C#

**Спецификация языка**

**Версия**

**Уведомление**

© Корпорация Майкрософт (Microsoft Corp.), 1999-2012. Все права защищены.

Microsoft, Windows, Visual Basic, Visual C# и Visual C++ являются охраняемыми товарными знаками корпорации Майкрософт в США и других странах.

Названия прочих организаций и изделий являются товарными знаками соответствующих владельцев.

**Содержание**

1. Введение 1

1.1 Программа «Hello world» 2

1.2 Структура программы 2

1.3 Типы и переменные 4

1.4 Выражения 8

1.5 Операторы 10

1.6 Классы и объекты 15

1.6.1 Члены 15

1.6.2 Специальные возможности 16

1.6.3 Параметры типа 16

1.6.4 Базовые классы 17

1.6.5 Поля 17

1.6.6 Методы 18

1.6.6.1 Параметры 18

1.6.6.2 Тело метода и локальные переменные 20

1.6.6.3 Статические методы и методы экземпляра 20

1.6.6.4 Виртуальные, переопределяющие и абстрактные методы 21

1.6.6.5 Перегрузка метода 24

1.6.7 Другие члены-функции 24

1.6.7.1 Конструкторы 26

1.6.7.2 Свойства 27

1.6.7.3 Индексаторы 27

1.6.7.4 События 28

1.6.7.5 Операторы 28

1.6.7.6 Деструкторы 29

1.7 Структуры 29

1.8 Массивы 31

1.9 Интерфейсы 32

1.10 Перечисления 33

1.11 Делегаты 35

1.12 Атрибуты 36

2. Лексическая структура 39

2.1 Программы 39

2.2 Грамматики 39

2.2.1 Грамматическая нотация 39

2.2.2 Лексика 40

2.2.3 Синтаксическая грамматика 40

2.3 Лексический анализ 40

2.3.1 Знаки завершения строки 41

2.3.2 Комментарии 42

2.3.3 Пробел 43

2.4 Лексемы 43

2.4.1 Управляющие последовательности символов Юникода 43

2.4.2 Идентификаторы 44

2.4.3 Ключевые слова 46

2.4.4 Литералы 47

2.4.4.1 Логические литералы 47

2.4.4.2 Целочисленные литералы 47

2.4.4.3 Действительные литералы 48

2.4.4.4 Символьные литералы 49

2.4.4.5 Строковые литералы 50

2.4.4.6 Литерал null 52

2.4.5 Операторы и знаки пунктуации 52

2.5 Препроцессорные директивы 52

2.5.1 Символы условной компиляции 54

2.5.2 Препроцессорные выражения 54

2.5.3 Директивы объявлений 55

2.5.4 Директивы условной компиляции 56

2.5.5 Директивы диагностики 58

2.5.6 Директивы областей 59

2.5.7 Директивы строк 59

2.5.8 Директивы pragma 60

2.5.8.1 Директива pragma warning 60

3. Основные понятия 63

3.1 Запуск приложения 63

3.2 Завершение приложения 64

3.3 Объявления 64

3.4 Члены 67

3.4.1 Члены пространства имен 67

3.4.2 Члены структуры 67

3.4.3 Члены перечисления 68

3.4.4 Члены класса 68

3.4.5 Члены интерфейса 68

3.4.6 Члены массива 68

3.4.7 Члены делегата 68

3.5 Доступ к члену 68

3.5.1 Объявленная доступность 68

3.5.2 Домены доступности 69

3.5.3 Защищенный доступ для членов экземпляров 72

3.5.4 Ограничения доступности 73

3.6 Сигнатуры и перегрузка 74

3.7 Области видимости 75

3.7.1 Скрытие имени. 78

3.7.1.1 Скрытие через вложение. 78

3.7.1.2 Скрытие через наследование 79

3.8 Имена пространств имен и типов 80

3.8.1 Полные имена 83

3.9 Автоматическое управление памятью 83

3.10 Порядок выполнения 86

4. Типы 87

4.1 Типы значений 87

4.1.1 Тип System.ValueType 88

4.1.2 Конструкторы по умолчанию 88

4.1.3 Типы структуры 89

4.1.4 Простые типы 89

4.1.5 Целочисленные типы 90

4.1.6 Типы с плавающей запятой 92

4.1.7 Тип decimal 93

4.1.8 Тип bool 94

4.1.9 Перечисляемые типы 94

4.1.10 Обнуляемые типы 94

4.2 Ссылочные типы 95

4.2.1 Типы классов 96

4.2.2 Тип объекта 96

4.2.3 Динамический тип 96

4.2.4 Строковый тип 96

4.2.5 Типы интерфейсов 96

4.2.6 Типы массивов 97

4.2.7 Типы делегатов 97

4.3 Упаковка и распаковка 97

4.3.1 Преобразования упаковки 97

4.3.2 Преобразования распаковки 99

4.4 Сформированные типы 100

4.4.1 Аргументы типа 100

4.4.2 Открытые и закрытые типы 101

4.4.3 Связанные и несвязанные типы 101

4.4.4 Соблюдение ограничений 101

4.5 Параметры типа 103

4.6 Типы дерева выражений 103

4.7 Динамический тип 104

5. Переменные 106

5.1 Категории переменных 106

5.1.1 Статические переменные 106

5.1.2 Переменные экземпляра 106

5.1.2.1 Переменные экземпляра в классах 107

5.1.2.2 Переменные экземпляра в структурах 107

5.1.3 Элементы массива 107

5.1.4 Параметры по значению 107

5.1.5 Параметры ссылок 107

5.1.6 Выходные параметры 108

5.1.7 Локальные переменные 108

5.2 Значения по умолчанию 109

5.3 Определенное присваивание 109

5.3.1 Переменные с начальным значением 110

5.3.2 Переменные без начального значения 111

5.3.3 Точные правила для выявления определенного присваивания 111

5.3.3.1 Общие правила для операторов 112

5.3.3.2 Операторы блока, операторы checked и unchecked 112

5.3.3.3 Операторы-выражения 112

5.3.3.4 Операторы объявления 112

5.3.3.5 Операторы If 112

5.3.3.6 Операторы switch 113

5.3.3.7 Операторы while 113

5.3.3.8 Операторы do 113

5.3.3.9 Операторы for 114

5.3.3.10 Операторы break, continue и goto 114

5.3.3.11 Операторы throw 114

5.3.3.12 Операторы return 114

5.3.3.13 Операторы try-catch 115

5.3.3.14 Операторы try-finally 115

5.3.3.15 Операторы try-catch-finally 115

5.3.3.16 Операторы foreach 116

5.3.3.17 Операторы using 116

5.3.3.18 Операторы lock 117

5.3.3.19 Операторы yield 117

5.3.3.20 Общие правила для простых выражений 117

5.3.3.21 Общие правила для выражений с внедренными выражениями 117

5.3.3.22 Выражения вызова и выражения создания объекта 118

5.3.3.23 Простые выражения присваивания 118

5.3.3.24 Выражения && 118

5.3.3.25 Выражения || 119

5.3.3.26 ! выражения 120

5.3.3.27 Выражения ?? 121

5.3.3.28 Выражения ?: 121

5.3.3.29 Анонимные функции 122

5.4 Ссылочные переменные 123

5.5 Атомарность ссылок на переменные 123

6. Преобразования 125

6.1 Неявные преобразования 125

6.1.1 Преобразование идентификатора 126

6.1.2 Неявные числовые преобразования 126

6.1.3 Неявные преобразования перечисляемых типов 126

6.1.4 Неявные преобразования обнуляемых типов 126

6.1.5 Преобразования литерала NULL. 127

6.1.6 Неявные преобразования ссылочных типов 127

6.1.7 Преобразования упаковки 128

6.1.8 Неявные преобразования динамических типов 128

6.1.9 Неявные преобразования выражений констант 129

6.1.10 Неявные преобразования, включающие параметры типа 129

6.1.11 Пользовательские неявные преобразования 129

6.1.12 Преобразования анонимных функций и преобразования группы методов 130

6.2 Явные преобразования 130

6.2.1 Явные преобразования числовых типов 130

6.2.2 Явные преобразования перечисляемых типов 132

6.2.3 Явные преобразования обнуляемых типов 132

6.2.4 Явные преобразования ссылочных типов 133

6.2.5 Преобразования распаковки 134

6.2.6 Неявные динамические преобразования. 134

6.2.7 Явные преобразования, включающие параметры типа 135

6.2.8 Пользовательские явные преобразования 136

6.3 Стандартные преобразования 136

6.3.1 Стандартные неявные преобразования 136

6.3.2 Стандартные явные преобразования 136

6.4 Пользовательские преобразования 137

6.4.1 Допустимые пользовательские преобразования 137

6.4.2 Операторы преобразования с нулификацией 137

6.4.3 Вычисление пользовательских преобразований 137

6.4.4 Определенные пользователем неявные преобразования 138

6.4.5 Пользовательские явные преобразования 139

6.5 Преобразования анонимных функций 140

6.5.1 Вычисление преобразований анонимных функций к типам делегата 142

6.5.2 Вычисление преобразования анонимной функции к типу дерева выражений 143

6.5.3 Пример реализации 143

6.6 Преобразования группы методов 145

7. Выражения 149

7.1 Классы выражений 149

7.1.1 Значения выражений 150

7.2 Статическая и динамическая привязка 150

7.2.1 Время привязки 151

7.2.2 Динамическая привязка 151

7.2.3 Типы составных выражений 152

7.3 Операторы 152

7.3.1 Приоритет и ассоциативность операторов 153

7.3.2 Перегрузка операторов 154

7.3.3 Разрешение перегрузки унарных операторов 155

7.3.4 Разрешение перегрузки бинарных операторов 155

7.3.5 Пользовательские операторы-кандидаты 156

7.3.6 Числовое расширение 156

7.3.6.1 Числовое расширение унарных операторов 157

7.3.6.2 Числовое расширение бинарных операторов 157

7.3.7 Операторы с нулификацией 158

7.4 Поиск членов 158

7.4.1 Базовые типы 160

7.5 Функции-члены 160

7.5.1 Списки аргументов 163

7.5.1.1 Соответствующие параметры 164

7.5.1.2 Вычисление списков аргументов во время выполнения 165

7.5.2 Вывод типа 167

7.5.2.1 Первый этап 168

7.5.2.2 Второй этап 168

7.5.2.3 Типы ввода 168

7.5.2.4 Типы вывода 168

7.5.2.5 Зависимость 169

7.5.2.6 Вывод типа вывода 169

7.5.2.7 Вывод явных типов параметров 169

7.5.2.8 Точный вывод 169

7.5.2.9 Вывод нижних границ 169

7.5.2.10 Вывод по верхним границам 170

7.5.2.11 Фиксирование 171

7.5.2.12 Выведенный тип возвращаемого значения 171

7.5.2.13 Вывод типа при преобразовании групп методов 172

7.5.2.14 Поиск наиболее подходящего общего типа для набора выражений 173

7.5.3 Разрешение перегрузки 173

7.5.3.1 Применимая функция-член 174

7.5.3.2 Более подходящая функция-член 174

7.5.3.3 Лучшее преобразование из выражения 176

7.5.3.4 Лучшее преобразование из типа 176

7.5.3.5 Лучшая цель для преобразования 176

7.5.3.6 Перегрузка в универсальных классах 177

7.5.4 Проверка динамического разрешения перегрузки во время компиляции 177

7.5.5 Вызов функции-члена 177

7.5.5.1 Вызов в упакованных экземплярах 179

7.6 Первичные выражения 179

7.6.1 Литералы 180

7.6.2 Простые имена 180

7.6.2.1 Инвариантность значения в блоках 181

7.6.3 Выражения со скобками 182

7.6.4 Доступ к члену 182

7.6.4.1 Идентичные простые имена и имена типов 185

7.6.4.2 Грамматическая неоднозначность 185

7.6.5 Выражения вызова 186

7.6.5.1 Вызовы методов 186

7.6.5.2 Вызовы методов расширения 188

7.6.5.3 Вызовы делегатов 190

7.6.6 Метод доступа к элементу 191

7.6.6.1 Доступ к массиву 191

7.6.6.2 Доступ к индексатору 192

7.6.7 Доступ this 193

7.6.8 Доступ base 193

7.6.9 Постфиксные операторы инкремента и декремента 194

7.6.10 Оператор new 195

7.6.10.1 Выражения создания объектов 195

7.6.10.2 Инициализаторы объектов 197

7.6.10.3 Инициализаторы коллекции 199

7.6.10.4 Выражения создания массива 200

7.6.10.5 Выражения создания делегата 202

7.6.10.6 Выражения создания анонимных объектов 204

7.6.11 Оператор typeof 205

7.6.12 Операторы checked и unchecked 207

7.6.13 Выражения значения по умолчанию 210

7.6.14 Выражения анонимного метода 210

7.7 Унарные операторы. 210

7.7.1 Унарный оператор «плюс» 210

7.7.2 Унарный оператор «минус» 211

7.7.3 Оператор логического отрицания 211

7.7.4 Оператор побитового дополнения 212

7.7.5 Префиксные операторы инкремента и декремента 212

7.7.6 Выражения приведения типа 213

7.7.7 Выражения await 214

7.7.7.1 Выражения с ожиданием 214

7.7.7.2 Классификация выражений await 215

7.7.7.3 Вычисление выражений await во время выполнения 215

7.8 Арифметические операторы 215

7.8.1 Оператор произведения 216

7.8.2 Оператор деления 217

7.8.3 Оператор остатка 218

7.8.4 Оператор сложения 219

7.8.5 Оператор вычитания 221

7.9 Операторы сдвига 222

7.10 Операторы отношения и проверки типа 224

7.10.1 Операторы сравнения целых чисел 225

7.10.2 Операторы сравнения чисел с плавающей запятой 225

7.10.3 Операторы сравнения десятичных чисел 226

7.10.4 Логические операторы равенства 226

7.10.5 Операторы сравнения значений перечисления 227

7.10.6 Операторы равенства значений ссылочного типа 227

7.10.7 Операторы равенства строк 229

7.10.8 Операторы равенства делегатов 229

7.10.9 Операторы равенства и значение NULL 230

7.10.10 Оператор is 230

7.10.11 Оператор as 230

7.11 Логические операторы 231

7.11.1 Логические операторы для целых чисел 232

7.11.2 Логические операторы для перечислений 232

7.11.3 Логические операторы 232

7.11.4 Обнуляемые логические операторы 233

7.12 Условные логические операторы 233

7.12.1 Логические условные операторы 234

7.12.2 Пользовательские условные логические операторы 234

7.13 Оператор слияния с NULL 235

7.14 Условный оператор 236

7.15 Выражения анонимных функций 236

7.15.1 Сигнатуры анонимных функций 238

7.15.2 Тела анонимных функций 239

7.15.3 Разрешение перегрузки 239

7.15.4 Анонимные функции и динамическая привязка 240

7.15.5 Внешние переменные 240

7.15.5.1 Захваченные внешние переменные 240

7.15.5.2 Создание экземпляров локальных переменных 241

7.15.6 Вычисление выражений анонимных функций 243

7.16 Выражения запросов 243

7.16.1 Неоднозначность в выражениях запросов 245

7.16.2 Перевод выражений запросов 245

7.16.2.1 Предложения select и groupby с продолжениями 245

7.16.2.2 Явные типы переменных диапазона 246

7.16.2.3 Выражения вырожденных запросов 246

7.16.2.4 Предложения from, let, where, join и orderby 247

7.16.2.5 Предложения select 250

7.16.2.6 Предложения groupby 251

7.16.2.7 Прозрачные идентификаторы 251

7.16.3 Шаблон выражения запроса 253

7.17 Операторы присваивания 254

7.17.1 Простое присваивание 254

7.17.2 Сложное присваивание 257

7.17.3 Присваивание событий 258

7.18 Выражение 258

7.19 Константные выражения 258

7.20 Логические выражения 260

8. Операторы 261

8.1 Конечные точки и достижимость 261

8.2 Блоки 263

8.2.1 Списки операторов 263

8.3 Пустой оператор 264

8.4 Помеченные операторы 264

8.5 Операторы объявления 265

8.5.1 Объявления локальных переменных 265

8.5.2 Объявления локальных констант 267

8.6 Операторы-выражения 267

8.7 Операторы выбора 268

8.7.1 Оператор if 268

8.7.2 Оператор switch 269

8.8 Операторы итераций 272

8.8.1 Оператор while 273

8.8.2 Оператор do 273

8.8.3 Оператор for 274

8.8.4 Оператор foreach 275

8.9 Операторы перехода 278

8.9.1 Оператор break 279

8.9.2 Оператор continue 280

8.9.3 Оператор goto 281

8.9.4 Оператор return 282

8.9.5 Оператор throw 283

8.10 Оператор try 284

8.11 Операторы checked и unchecked 287

8.12 Оператор lock 287

8.13 Оператор using 288

8.14 Оператор yield 290

9. Пространства имен 292

9.1 Единицы компиляции 292

9.2 Объявления пространства имен 292

9.3 Внешние псевдонимы 294

9.4 Директивы using 294

9.4.1 Директивы using alias 295

9.4.2 Директивы using namespace 298

9.5 Члены пространства имен 299

9.6 Объявления типов 300

9.7 Квалификаторы псевдонима пространства имен 300

9.7.1 Уникальность псевдонимов 301

10. Классы 303

10.1 Объявления классов 303

10.1.1 Модификаторы классов 303

10.1.1.1 Абстрактные классы 304

10.1.1.2 Запечатанные классы 304

10.1.1.3 Статические классы 305

10.1.2 Модификатор partial 305

10.1.3 Параметры типа 306

10.1.4 Спецификация базы класса 306

10.1.4.1 Базовые классы 306

10.1.4.2 Реализация интерфейсов 308

10.1.5 Ограничения параметров типа 308

10.1.6 Тело класса 313

10.2 Разделяемые типы 313

10.2.1 Атрибуты 313

10.2.2 Модификаторы 314

10.2.3 Параметры и ограничения типа 314

10.2.4 Базовый класс 315

10.2.5 Базовые интерфейсы 315

10.2.6 Члены 315

10.2.7 Разделяемые методы 316

10.2.8 Привязка имен 318

10.3 Члены класса 319

10.3.1 Тип экземпляра 320

10.3.2 Члены сформированных типов 321

10.3.3 Наследование 322

10.3.4 Модификатор new 323

10.3.5 Модификаторы доступа 323

10.3.6 Составные типы 323

10.3.7 Статические члены и члены экземпляра 323

10.3.8 Вложенные типы 324

10.3.8.1 Полные имена 324

10.3.8.2 Объявленная доступность 325

10.3.8.3 Скрытие 325

10.3.8.4 Доступ this 326

10.3.8.5 Доступ к частным и защищенным членам типа-контейнера 327

10.3.8.6 Вложенные типы в универсальных классах 328

10.3.9 Зарезервированные имена членов 328

10.3.9.1 Имена членов, зарезервированные для свойств 329

10.3.9.2 Имена членов, зарезервированные для событий 329

10.3.9.3 Имена членов, зарезервированные для индексаторов 329

10.3.9.4 Имена членов, зарезервированные для деструкторов 330

10.4 Константы 330

10.5 Поля 331

10.5.1 Статические поля и поля экземпляров 333

10.5.2 Поля только для чтения 333

10.5.2.1 Использование статических полей только для чтения вместо констант 334

10.5.2.2 Отслеживание версий констант и статических полей только для чтения 334

10.5.3 Поля с модификатором volatile 335

10.5.4 Инициализация поля 336

10.5.5 Инициализаторы переменных 336

10.5.5.1 Инициализация статического поля 337

10.5.5.2 Инициализация поля экземпляра 339

10.6 Методы 339

10.6.1 Параметры метода 341

10.6.1.1 Параметры по значению 343

10.6.1.2 Параметры ссылок 343

10.6.1.3 Выходные параметры 344

10.6.1.4 Массивыпараметров 345

10.6.2 Статические методы и методы экземпляра 348

10.6.3 Виртуальные методы 348

10.6.4 Переопределяющие методы 350

10.6.5 Запечатанные методы 352

10.6.6 Абстрактные методы 353

10.6.7 Внешние методы 354

10.6.8 Разделяемые методы 355

10.6.9 Методы расширения 355

10.6.10 Тело метода 356

10.6.11 Перегрузка метода 357

10.7 Свойства 357

10.7.1 Статические свойства и свойства экземпляра 358

10.7.2 Методы доступа 358

10.7.3 Автоматически реализуемые свойства 364

10.7.4 Специальные возможности 364

10.7.5 Виртуальные, запечатанные, переопределяющие и абстрактные методы доступа 365

10.8 События 367

10.8.1 События, подобные полям 369

10.8.2 Методы доступа к событиям 370

10.8.3 Статические события и события экземпляров 372

10.8.4 Виртуальные, запечатанные, переопределяющие и абстрактные методы доступа 372

10.9 Индексаторы 373

10.9.1 Перегрузка индексатора 376

10.10 Операторы 376

10.10.1 Унарные операторы. 378

10.10.2 Бинарные операторы 379

10.10.3 Операторы преобразования 379

10.11 Конструкторы экземпляров 382

10.11.1 Инициализаторы конструкторов 383

10.11.2 Инициализаторы переменных экземпляров 383

10.11.3 Выполнение конструктора 384

10.11.4 Конструкторы по умолчанию 385

10.11.5 Закрытые конструкторы 386

10.11.6 Необязательные параметры конструктора экземпляров 386

10.12 Статические конструкторы 387

10.13 Деструкторы 389

10.14 Итераторы 391

10.14.1 Интерфейсы перечислителя 391

10.14.2 Перечислимые интерфейсы 391

10.14.3 Тип yield 391

10.14.4 Объекты перечислителя 391

10.14.4.1 Метод MoveNext 392

10.14.4.2 Свойство Current 393

10.14.4.3 Метод Dispose 393

10.14.5 Перечислимые объекты 394

10.14.5.1 Метод GetEnumerator 394

10.14.6 Пример реализации 394

10.15 Асинхронные функции 400

10.15.1 Вычисление асинхронной функции, возвращающей задачу 401

10.15.2 Вычисление асинхронной функции, возвращающей значение void 401

11. Структуры 402

11.1 Объявления структур 402

11.1.1 Модификаторы структуры 402

11.1.2 Модификатор partial 403

11.1.3 Интерфейсы структуры 403

11.1.4 Тело структуры 403

11.2 Члены структуры 403

11.3 Различия между классом и структурой 403

11.3.1 Семантика значений 404

11.3.2 Наследование 405

11.3.3 Присваивание 405

11.3.4 Значения по умолчанию 405

11.3.5 Упаковка и распаковка 406

11.3.6 Действие ключевого слова this 408

11.3.7 Инициализаторы полей 408

11.3.8 Конструкторы 408

11.3.9 Деструкторы 409

11.3.10 Статические конструкторы 409

11.4 Примеры структур 409

11.4.1 Тип целочисленного значения в базе данных 410

11.4.2 Логический тип базы данных 411

12. Массивы 415

12.1 Типы массива 415

12.1.1 Тип System.Array 416

12.1.2 Массивы и универсальный интерфейс IList 416

12.2 Создание массива 417

12.3 Доступ к элементам массива 417

12.4 Члены массива 417

12.5 Ковариация массивов 417

12.6 Инициализаторы массива 418

13. Интерфейсы 421

13.1 Объявления интерфейсов 421

13.1.1 Модификаторы интерфейса 421

13.1.2 Модификатор partial 421

13.1.3 Списки параметров типа варианта 422

13.1.3.1 Безопасность вариативности 422

13.1.3.2 Вариантные преобразования 423

13.1.4 Базовые интерфейсы 423

13.1.5 Тело интерфейса 424

13.2 Члены интерфейса 424

13.2.1 Методы интерфейса 425

13.2.2 Свойства интерфейса 426

13.2.3 События интерфейса 426

13.2.4 Индексаторы интерфейса 426

13.2.5 Доступ к членам интерфейса 427

13.3 Полные имена членов интерфейса 429

13.4 Реализация интерфейсов 429

13.4.1 Явные реализации членов интерфейса 430

13.4.2 Уникальность реализованных интерфейсов 432

13.4.3 Реализация универсальных методов 433

13.4.4 Сопоставление интерфейсов 434

13.4.5 Наследование реализаций интерфейсов 437

13.4.6 Повторная реализация интерфейса 438

13.4.7 Абстрактные классы и интерфейсы 439

14. Перечисления 441

14.1 Объявления перечислений 441

14.2 Модификаторы перечисления 442

14.3 Члены перечисления 442

14.4 Тип System.Enum 444

14.5 Значения перечисления и операции 444

15. Делегаты 447

15.1 Объявления делегатов 447

15.2 Совместимость делегатов 449

15.3 Создание экземпляра делегата 450

15.4 Вызов делегата 450

16. Исключения 453

16.1 Причины исключений 453

16.2 Класс System.Exception 453

16.3 Обработка исключений 453

16.4 Общие классы исключений 454

17. Атрибуты 457

17.1 Классы атрибутов 457

17.1.1 Использование атрибутов 457

17.1.2 Позиционные и именованные параметры 458

17.1.3 Типы параметров атрибута 459

17.2 Спецификация атрибута 459

17.3 Экземпляры атрибутов 465

17.3.1 Компиляция атрибута 465

17.3.2 Извлечение экземпляра атрибута во время выполнения 465

17.4 Зарезервированные атрибуты 466

17.4.1 Атрибут AttributeUsage 466

17.4.2 Атрибут Conditional 467

17.4.2.1 Условные методы 467

17.4.2.2 Классы условных атрибутов 469

17.4.3 Атрибут Obsolete 470

17.4.4 Атрибуты сведений о вызывающем коде 471

17.4.4.1 Атрибут CallerLineNumber 472

17.4.4.2 Атрибут CallerFilePath 472

17.4.4.3 Атрибут CallerMemberName 473

17.5 Атрибуты для взаимодействия 473

17.5.1 Взаимодействие с компонентами COM и Win32 473

17.5.2 Взаимодействие с другими языками .NET 473

17.5.2.1 Атрибут IndexerName 473

18. Небезопасный код 475

18.1 Небезопасные контексты 475

18.2 Типы указателей 478

18.3 Фиксированные и перемещаемые переменные 480

18.4 Преобразования указателей 481

18.4.1 Массивы указателей 482

18.5 Указатели в выражениях 483

18.5.1 Косвенное обращение по указателю 483

18.5.2 Доступ к члену по указателю 484

18.5.3 Доступ к элементу по указателю 485

18.5.4 Оператор адреса 485

18.5.5 Увеличение и уменьшение указателя 486

18.5.6 Арифметические операции с указателем 487

18.5.7 Сравнение указателей 487

18.5.8 Оператор sizeof 488

18.6 Оператор fixed 488

18.7 Буферы фиксированного размера 492

18.7.1 Объявления буферов фиксированного размера 492

18.7.2 Буферы фиксированного размера в выражениях 494

18.7.3 Проверка определенного присваивания 495

18.8 Выделение стека 495

18.9 Динамическое выделение памяти 496

A. Комментарии к документации 499

A.1 Введение 499

A.2 Рекомендованные теги 500

A.2.1 <c> 501

A.2.2 <code> 502

A.2.3 <example> 502

A.2.4 <exception> 502

A.2.5 <include> 503

A.2.6 <list> 504

A.2.7 <para> 504

A.2.8 <param> 505

A.2.9 <paramref> 505

A.2.10 <permission> 506

A.2.11 <remark> 506

A.2.12 <returns> 506

A.2.13 <see> 507

A.2.14 <seealso> 507

A.2.15 <summary> 508

A.2.16 <value> 508

A.2.17 <typeparam> 508

A.2.18 <typeparamref> 509

A.3 Обработка файла документации 509

A.3.1 Формат строки идентификатора 509

A.3.2 Примеры строк идентификаторов 511

A.4 Пример 514

A.4.1 Исходный код C# 514

A.4.2 Результирующий XML 516

B. Грамматика 522

B.1 Лексика 522

B.1.1 Знаки завершения строки 522

B.1.2 Комментарии 522

B.1.3 Пробел 523

B.1.4 Лексемы 523

B.1.5 Управляющие последовательности символов Юникода 523

B.1.6 Идентификаторы 523

B.1.7 Ключевые слова 524

B.1.8 Литералы 525

B.1.9 Операторы и знаки пунктуации 527

B.1.10 Директивы предварительной обработки 527

B.2 Синтаксис 529

B.2.1 Основные понятия 529

B.2.2 Типы 529

B.2.3 Переменные 531

B.2.4 Выражения 531

B.2.5 Операторы 538

B.2.6 Пространства имен 541

B.2.7 Классы 542

B.2.8 Структуры 549

B.2.9 Массивы 550

B.2.10 Интерфейсы 550

B.2.11 Перечисления 551

B.2.12 Делегаты 552

B.2.13 Атрибуты 552

B.3 Грамматические расширения для небезопасного кода 554

C. Ссылки 557

# Введение

C# (произносится как «Си-шарп» или «Си-диез») — это простой, современный, типобезопасный объектно-ориентированный язык программирования. C# базируется на семействе языков программирования C и будет хорошо знаком программистам, работавшим с языками C, C++ и Java. Язык C# стандартизирован организацией ECMA International на соответствие стандарту ECMA-334, а также организацией ISO/IEC на соответствие стандарту ISO/IEC 23270. Компилятор Microsoft C# для платформы .NET Framework реализован в полном соответствии с требованиями обоих стандартов.

C# представляет собой объектно-ориентированный язык программирования, однако также обеспечивает поддержку компонентно-ориентированного программирования. Разработка современных приложений все в большей степени базируется на применении программных компонентов в форме автономных и самодокументируемых функциональных модулей. Основной особенностью таких компонентов является реализация модели программирования с использованием свойств, методов, событий и атрибутов, представляющих декларативное описание компонентов, а также включение в них собственной документации. В C# представлены языковые конструкции, непосредственно поддерживающие эти понятия, что делает его близким к естественному языком для создания и применения программных компонентов.

В C# представлены функциональные возможности, позволяющие создавать надежные и устойчивые приложения. Среди них: функция сборки мусора для автоматического освобождения памяти, занимаемой неиспользуемыми объектами; функция обработки исключений, обеспечивающая структурированный и расширяемый подход к обнаружению и устранению ошибок; а также строго типизированная структура языка, не допускающая считывания неинициализированных переменных, выхода индекса массива за пределы допустимого диапазона или выполнения непроверенных приведений типов.

В C# применяется унифицированная система типов. Все типы C#, включая типы-примитивы (например, int и double), наследуются от единственного корневого типа object. Таким образом, все типы используют набор общих операций, что обеспечивает согласованные хранение, передачу и обработку значений любого типа. Кроме того, в C# поддерживаются пользовательские ссылочные типы и типы значений, что обеспечивает динамическое размещение объектов в памяти и встроенное хранение упрощенных структур.

Чтобы обеспечить совместимость и возможность дальнейшего развития программ и библиотек C#, в языке C# большое внимание уделяется управлению версиями. В большинстве языков программирования этому вопросу уделяется недостаточное внимание, в результате чего в создаваемых на таких языках программах чаще обычного возникают проблемы при переходе на новые версии зависимых библиотек. В C# реализованы следующие возможности по управлению версиями: разделение модификаторов virtual и override, применение правил разрешения перегрузки метода и поддержка явного объявления членов интерфейса.

В данной главе описываются важнейшие функциональные возможности языка C#. В последующих главах правила и исключения описываются более детально и зачастую с математическими подробностями. Однако в этой главе акцент сделан на четкости и сжатости, а не на полноте изложения. Основная цель данной главы — предоставить вводную информацию о языке, которая будет полезна при написании простых программ, а также при чтении последующих глав.

## Программа «Hello world»

Программа «Hello, World» традиционно используется для знакомства с языком программирования. Ниже приведен пример этой программы на языке C#:

using System;

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

Исходные файлы C# обычно имеют расширение .cs. Если программа "Hello, World" сохранена в файле hello.cs, то чтобы откомпилировать ее с помощью компилятора Microsoft C#, введите следующую команду в командной строке:

csc hello.cs

В результате компиляции создается исполняемый файл сборки hello.exe. В результате выполнения этого приложения получаются следующие выходные данные:

Hello, World

Программа "Hello, World" начинается с директивы using, которая ссылается на пространство имен System. Пространства имен используются для иерархического упорядочения программ и библиотек C#. Пространства имен могут содержать типы и другие пространства имен. Например, пространство имен System содержит набор типов (например, используемый в программе класс Console), а также ряд других пространств имен (например, IO и Collections). Директива using ссылается на заданное пространство имен и обеспечивает возможность использования неполных имен типов, являющихся его членами. Благодаря применению директивы using в программе можно использовать сокращенную форму записи Console.WriteLine вместо System.Console.WriteLine.

Класс Hello, объявленный в программе "Hello, World", содержит единственный член — метод Main. Метод Main объявляется с помощью модификатора static. В C# методы могут ссылаться на конкретный экземпляр содержащего объекта с помощью ключевого слова this, однако статические методы работают без ссылки на конкретный объект. По соглашению статический метод Main является точкой входа в программу.

Выходные данные программы формируются с помощью метода WriteLine класса Console, принадлежащего пространству имен System. Этот класс содержится в библиотеках классов платформы .NET Framework, на которые по умолчанию автоматически ссылается компилятор Microsoft C#. Обратите внимание, что в C# не используется собственная библиотека времени выполнения. Вместо этого в C# в качестве библиотеки времени выполнения используется платформа .NET Framework.

## Структура программы

В C# используются следующие основные структурные понятия: программа, пространство имен, тип, член и сборка. Программа C# состоит из одного или нескольких исходных файлов. В программе объявляются типы, которые содержат члены и могут быть упорядочены в пространствах имен. Примерами типов являются классы и интерфейсы. Примерами членов являются поля, методы, свойства и события. При компиляции программ C# выполняется их физическая упаковка в сборки. Файлы сборок обычно имеют расширение .exe или .dll и представляют собой приложения или библиотеки соответственно.

Пример:

using System;

namespace Acme.Collections  
{  
 public class Stack  
 {  
 Entry top;

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;  
 }  
 }  
 }  
}

объявляется класс Stack, принадлежащий пространству имен Acme.Collections. Полное имя этого класса — Acme.Collections.Stack. Этот класс содержит несколько членов: поле top, два метода с именами Push и Pop, а также вложенный класс Entry. Класс Entry, в свою очередь, содержит три члена: поля с именами next и data, а также конструктор. Если исходный код примера хранится в файле acme.cs, команда

csc /t:library acme.cs

компилирует пример в виде библиотеки (код без точки входа Main) и создает сборку с именем acme.dll.

Сборки содержат исполняемый код в форме инструкций промежуточного языка (Intermediate Language, IL), а также символьные данные в форме метаданных. Перед выполнением код IL сборки автоматически преобразуется в код для конкретного процессора с помощью JIT-компилятора среды .NET CLR.

Поскольку сборка представляет собой самодокументируемый функциональный модуль, содержащий как код, так и метаданные, в файлах заголовка C# не требуется использовать директивы #include. В программе C# возможность обращения к открытым типам и членам, содержащимся в конкретной сборке, реализуется посредством ссылки на эту сборку во время компиляции программы. Например, в следующей программе используется класс Acme.Collections.Stack, содержащийся в сборке acme.dll:

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());  
 }  
}

Если программа хранится в файле test.cs, во время его компиляции test.cs ссылка на сборку acme.dll реализуется с помощью параметра компилятора /r:

csc /r:acme.dll test.cs

В результате этого создается исполняемый файл сборки test.exe, при выполнении которого получаются следующие выходные данные:

100  
10  
1

В C# допускается хранение исходного текста программы в нескольких исходных файлах. При компиляции многофайловой программы C# выполняется совместная обработка всех исходных файлов. Это позволяет использовать в каждом файле ссылки на другие файлы. Концептуально такой подход соответствует объединению всех файлов перед обработкой в единый исходный файл. В C# никогда не используются опережающие объявления, поскольку, за редким исключением, порядок объявления не имеет значения. В C# исходный файл необязательно должен содержать объявление только одного открытого типа. Также имя исходного файла необязательно должно совпадать с именем объявленного в нем типа.

## Типы и переменные

В C# все типы подразделяются на две основные категории: типы значений и ссылочные типы. Переменные типа значений непосредственно содержат данные, тогда как переменные ссылочного типа хранят ссылки на соответствующие данные (объекты). Две переменные ссылочного типа могут ссылаться на один объект. Это позволяет изменять объект, на который ссылается одна переменная, выполняя соответствующие операции с другой. Каждая переменная типа значений содержит собственную копию данных. В связи с этим операции с одной переменной не влияют на другую (за исключением переменных параметров ref и out).

Типы значений в C# подразделяются на простые типы, перечисляемые типы, типы структур и обнуляемые типы. Ссылочные типы в C# подразделяются на типы классов, типы интерфейсов, типы массивов и типы делегатов.

В следующей таблице представлен обзор системы типов C#.

|  |  |  |
| --- | --- | --- |
| **Категория** | | **Описание** |
| Типы значений | Простые типы | Целые со знаком: sbyte, short, int, long |
| Целые без знака: byte, ushort, uint, ulong |
| Символы Юникода: char |
| Числа плавающей запятой (стандарт IEEE): float, double |
| Десятичный повышенной точности: decimal |
| Логические значения: bool |
| Перечисляемые типы | Пользовательские типы в формате enum E {...} |
| Типы структуры | Пользовательские типы в формате struct S {...} |
| Обнуляемые типы | Расширения любых других типов значений, допускающие значение null |
| Ссылочные типы | Типы классов | Первичный базовый класс для всех типов: object |
| Строки Юникода: string |
| Пользовательские типы в формате class C {...} |
| Типы интерфейсов | Пользовательские типы в формате interface I {...} |
| Типы массива | Одномерные и многомерные, например int[] и int[,] |
| Типы делегатов | Пользовательские типы вида delegate int D(...) |

Восемь целых типов обеспечивают поддержку 8-, 16-, 32- и 64-разрядных значений со знаком и без знака.

Два типа с плавающей запятой (float и double) представляются в 32-битовом (одинарная точность) или 64-битовом (двойная точность) формате IEEE 754.

Тип decimal представляет собой 128-битовый тип данных, ориентированный на применение в финансовых и денежных вычислениях.

Тип bool в языке C# используется для представления логических значений — true и false.

Обработка знаков и строк в C# выполняется с применением кодировки Юникод. Тип char представляет элемент кодировки UTF-16, а тип string — последовательность элементов кодировки UTF-16.

В следующей таблице представлено краткое описание числовых типов C#.

|  |  |  |  |
| --- | --- | --- | --- |
| **Категория** | **Разрядность** | **Тип** | **Диапазон и точность** |
| Целые со знаком | 8 | sbyte | –128...127 |
| 16 | short | –32,768...32,767 |
| 32 | int | –2,147,483,648...2,147,483,647 |
| 64 | long | –9,223,372,036,854,775,808...9,223,372,036,854,775,807 |
| Целые без знака | 8 | byte | 0...255 |
| 16 | ushort | 0...65,535 |
| 32 | uint | 0...4,294,967,295 |
| 64 | ulong | 0...18,446,744,073,709,551,615 |
| С плавающей запятой | 32 | float | От 1,5 × 10−45 до 3,4 × 1038 с точностью до 7 знаков |
| 64 | double | От 5,0 × 10−324 до 1,7 × 10308 с точностью до 15 знаков |
| Десятичный | 128 | decimal | От 1,0 × 10−28 до 7,9 × 1028 с точностью до 28 знаков |

Для создания новых типов в программах C# используются объявления типов. В объявлении типа указываются имя и члены нового типа. В C# поддерживается определение пользователем типов пяти категорий: типы класса, типы структуры, типы интерфейса, перечисляемые типы и типы делегата.

Тип класса определяет структуру данных, содержащую данные-члены (поля) и функции-члены (методы, свойства и другие). Типы класса поддерживают механизмы единичного наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие функциональные возможности базового класса.

Тип структуры, как и тип класса, представляет структуру, содержащую данные-члены и функции-члены. Однако структуры, в отличие от классов, являются типами значений и для них не требуется выделение памяти в куче. Типы структуры не поддерживают определяемое пользователем наследование. Все типы структуры неявно наследуются от типа object.

Тип интерфейса определяет контракт как именованный набор открытых функций-членов. Класс или структура, реализующие интерфейс, должны предусматривать реализацию функций-членов интерфейса. Интерфейс может наследовать от нескольких базовых интерфейсов, а в классе или структуре может быть реализовано несколько интерфейсов.

Тип делегата представляет ссылки на методы с конкретным списком параметров и типом возвращаемого значения. С помощью делегатов методы обрабатываются как сущности, которым можно передавать параметры и которые можно присваивать переменным. Понятие делегата близко к понятию указателя на функцию, используемому в некоторых других языках. Однако делегаты, в отличие от указателей на функции, представляют собой пример объектно-ориентированного и типобезопасного подхода к программированию.

Типы класса, структуры, интерфейса и делегата поддерживают универсальные шаблоны, с помощью которых они могут быть параметризованы с другими типами.

Перечисляемые типы содержат конечное число значений, представляющих собой именованные константы. Для каждого перечисляемого типа существует базовый тип, в качестве которого используется один из восьми целых типов. Набор значений перечисляемого типа соответствует набору значений базового типа.

В C# поддерживаются одно- и многомерные массивы любого типа. В отличие от типов, перечисленных выше, типы массива необязательно объявлять перед использованием. Чтобы сформировать тип массива, необходимо ввести квадратные скобки после имени массива. Например, int[] — это одномерный массив типа int; int[,] — двумерный массив типа int, а int[][] — одномерный массив, состоящий из одномерных массивов типа int.

Обнуляемые типы также не требуется объявлять перед использованием. Для каждого необнуляемого типа значений T существует соответствующий обнуляемый тип T?, который дополнительно содержит значение null. Например, тип int? может содержать любое 32-разрядное целое значение, а также значение null.

В C# применяется унифицированная система типов, в которой значение любого типа может обрабатываться как объект. В C# каждый тип прямо или косвенно наследует от типа класса object, и класс object является первичным базовым классом для всех типов. Обработка значений ссылочного типа как объектов выполняется посредством рассмотрения их как значений типа object. Значения типа значений обрабатываются как объекты с использованием операций упаковки и распаковки. В следующем примере значение типа int преобразуется к типу object и затем обратно к типу int.

using System;

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

При преобразовании значения типа значений к типу object выделяется экземпляр объекта (также называется "бокс"), который будет содержать значение, после чего значение копируется в этот экземпляр. При обратном приведении ссылки на object к типу значений проверяется, является ли объект, на который существует ссылка, "боксом" допустимого типа значений. В случае успешного прохождения проверки копируется значение из "бокса".

В унифицированной системе типов C# типы значений могут преобразовываться в объекты "по требованию". Благодаря унификации как ссылочные типы, так и типы значений могут использовать универсальные библиотеки, использующие тип object.

В C# используются переменные различных типов, в том числе поля, элементы массива, локальные переменные и параметры. Переменная представляет собой ячейку для хранения значений. Каждой переменной присваивается тип, определяющий, какие значения могут храниться в ней, как показано в следующей таблице.

|  |  |
| --- | --- |
| **Тип переменной** | **Возможное содержимое** |
| Необнуляемый тип значений | Значение такого же типа |
| Обнуляемый тип значений | Значение null или значение такого же типа |
| object | Пустая ссылка, ссылка на объект любого ссылочного типа или ссылка на упакованное значение любого типа значений |
| Тип класса | Пустая ссылка, ссылка на экземпляр заданного типа класса или ссылка на экземпляр класса, производного от заданного типа |
| Тип интерфейса | Пустая ссылка, ссылка на экземпляр типа класса, который реализует заданный тип интерфейса, или ссылка на упакованное значение типа значений, который реализует заданный тип интерфейса |
| Тип массива | Пустая ссылка, ссылка на экземпляр заданного типа массива или ссылка на экземпляр совместимого типа массива |
| Тип делегата | Пустая ссылка или ссылка на экземпляр заданного типа делегата |

## Выражения

Выражения состоят из операндов и операторов. Операторы в выражении указывают, какие операции производятся с операндами. К операторам относятся, например, +, -, \*, / и new. К операндам относятся, например, литералы, поля, локальные переменные и выражения.

Когда выражение содержит несколько операторов, порядок вычисления отдельных операторов задается приоритетом операторов. Например, выражение x + y \* z вычисляется как x + (y \* z), поскольку оператор \* имеет более высокий приоритет по сравнению с оператором +.

Большинство операторов могут быть перегружены. Перегрузка операторов позволяет задавать пользовательские реализации операторов, в которых один или несколько операндов имеют пользовательский тип класса или структуры.

В следующей таблице приводится описание операторов C# в порядке убывания приоритета соответствующих категорий операторов. Операторы одной категории обладают одинаковым приоритетом.

|  |  |  |
| --- | --- | --- |
| **Категория** | **Выражение** | **Описание** |
| Основной | x.m | Доступ к члену |
| x(...) | Вызов методов и делегатов |
| x[...] | Доступ к массиву и индексатору |
| x++ | Постфиксный инкремент |
| x-- | Постфиксный декремент |
| new T(...) | Создание объекта и делегата |
| new T(...){...} | Создание объекта с использованием инициализатора |
| new {...} | Инициализатор анонимного объекта |
| new T[...] | Создание массива |
| typeof(T) | Получение объекта System.Type для T |
| checked(x) | Вычисление выражения в контексте checked |
| unchecked(x) | Вычисление выражения в контексте unchecked |
| default(T) | Получение значения по умолчанию типа T |
| delegate {...} | Анонимная функция (анонимный метод) |
| Унарный | +x | Идентификация |
| -x | Отрицание |
| !x | Логическое отрицание |
| ~x | Побитовое отрицание |
| ++x | Префиксный инкремент |
| --x | Префиксный декремент |
| (T)x | Явное преобразование x к типу T |
| await x | Асинхронное ожидание завершения x |
| Мультипликативный | x \* y | Умножение |
| x / y | Деление |
| x % y | Остаток |

|  |  |  |
| --- | --- | --- |
| Аддитивный | x + y | Сложение, объединение строк, объединение делегатов |
| x – y | Вычитание, удаление делегата |
| Сдвиг | x << y | Поразрядный сдвиг влево |
| x >> y | Поразрядный сдвиг вправо |
| Отношение и проверка типа | x < y | Меньше |
| x > y | Больше |
| x <= y | Меньше или равно |
| x >= y | Больше или равно |
| x is T | Возвращает значение true, если x относится к типу T, в противном случае — значение false |
| x as T | Возвращает значение x, типизированное как T, или значение null, если x относится к типу, отличному от T |
| Равенство | x == y | Равно |
| x != y | Не равно |
| Логическое И | x & y | Целое побитовое И, логическое И |
| Исключающее ИЛИ | x ^ y | Целое побитовое исключающее ИЛИ, логическое исключающее ИЛИ |
| Логическое ИЛИ | x | y | Целое побитовое ИЛИ, логическое ИЛИ |
| Условное И | x && y | Вычисляет значение y, только если значение x равно true |
| Условное ИЛИ | x || y | y вычисляется только в том случае, если x имеет значение false |
| Объединение с нулем | X ?? y | Вычисление возвращает значение y, если x имеет значение null; в противном случае вычисляется значение x |
| Условный | x ? y : z | Вычисляет значение y, если значение x равно true, и значение z, если значение x равно false |
| Присваивание или анонимная функция | x = y | Присваивание |
| x op= y | Составное присваивание; поддерживаются следующие операторы:  \*= /= %= += -= <<= >>= &= ^= |= |
| (T x) => y | Анонимная функция (лямбда-выражение) |

## Операторы

Действия программы выражаются с помощью операторов. В C# поддерживается несколько различных типов операторов, некоторые из которых определяются как внедренные операторы.

Блок позволяет записать несколько операторов языка в контексте, обычно допускающем использование только одного оператора. Блок состоит из нескольких операторов, записанных между разделителями — { и }.

Операторы объявления используются для объявления локальных переменных и констант.

Операторы выражений используются для вычисления выражений. Как операторы могут использоваться следующие выражения: вызовы методов, выделение объектов с помощью оператора new, операторы присваивания, использующие оператор =, и составного присваивания, операции приращения и уменьшения с помощью операторов ++ и --, а также выражения await.

Операторы выбора используются для выполнения одного из нескольких возможных операторов на основании значения какого-либо выражения. В этой группе представлены операторы if и switch.

Операторы итераций используются для повторяющегося выполнения внедренного оператора. В этой группе представлены операторы while, do, for и foreach.

Операторы перехода используются для передачи управления. В этой группе представлены операторы break, continue, goto, throw, return и yield.

Оператор try...catch используется для перехвата исключений, происходящих во время выполнения блока. Оператор try...finally используется для задания кода финализации, который выполняется всегда (независимо от того, возникло или нет исключение).

Операторы checked и unchecked используются для управления контекстом контроля переполнения в арифметических операциях и преобразованиях целых типов.

Оператор lock используется для получения блокировки взаимного исключения для заданного объекта, выполнения оператора и последующего снятия блокировки.

Оператор using используется для получения ресурса, выполнения оператора и последующего удаления ресурса.

В следующей таблице представлены операторы C# и примеры их использования.

|  |  |
| --- | --- |
| **Оператор** | **Пример** |
| Объявление локальной переменной | static void Main() {  int a;   int b = 2, c = 3;   a = 1;  Console.WriteLine(a + b + c); } |
| Объявление локальной константы | static void Main() {  const float pi = 3.1415927f;  const int r = 25;  Console.WriteLine(pi \* r \* r); } |
| Оператор выражения | static void Main() {  int i;  i = 123; // Expression statement  Console.WriteLine(i); // Expression statement  i++; // Expression statement  Console.WriteLine(i); // Expression statement } |
| Оператор if | static void Main(string[] args) {  if (args.Length == 0) {  Console.WriteLine("No arguments");  }  else {  Console.WriteLine("One or more arguments");  } } |

|  |  |
| --- | --- |
| Оператор switch | 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;  }  } } |
| Оператор while | static void Main(string[] args) {  int i = 0;  while (i < args.Length) {  Console.WriteLine(args[i]);  i++;  } } |
| Оператор do | static void Main() {  string s;  do {  s = Console.ReadLine();  if (s != null) Console.WriteLine(s);  } while (s != null); } |
| Оператор for | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  Console.WriteLine(args[i]);  } } |
| Оператор foreach | static void Main(string[] args) {  foreach (string s in args) {  Console.WriteLine(s);  } } |
| Оператор break | static void Main() {  while (true) {  string s = Console.ReadLine();  if (s == null) break;  Console.WriteLine(s);  } } |
| Оператор continue | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  if (args[i].StartsWith("/")) continue;  Console.WriteLine(args[i]);  } } |

|  |  |
| --- | --- |
| Оператор goto | static void Main(string[] args) {  int i = 0;  goto check;  loop:  Console.WriteLine(args[i++]);  check:  if (i < args.Length) goto loop; } |
| Оператор return | static int Add(int a, int b) {  return a + b; }  static void Main() {  Console.WriteLine(Add(1, 2));  return; } |
| Оператор yield | static IEnumerable<int> Range(int from, int to) {  for (int i = from; i < to; i++) {  yield return i;  }  yield break; }  static void Main() {  foreach (int x in Range(-10,10)) {  Console.WriteLine(x);  } } |
| Операторы throw и try | 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!”);  } } |
| Операторы checked и unchecked | static void Main() {  int i = int.MaxValue;  checked {  Console.WriteLine(i + 1); // Exception  }  unchecked {  Console.WriteLine(i + 1); // Overflow  } } |

|  |  |
| --- | --- |
| Оператор lock | class Account {  decimal balance;  public void Withdraw(decimal amount) {  lock (this) {  if (amount > balance) {  throw new Exception("Insufficient funds");  }  balance -= amount;  }  } } |
| Оператор using | static void Main() {  using (TextWriter w = File.CreateText("test.txt")) {  w.WriteLine("Line one");  w.WriteLine("Line two");  w.WriteLine("Line three");  } } |

## Классы и объекты

В C# классы представляют собой фундаментальные типы. Класс — это структура данных, объединяющая состояние (поля) и действия (методы и другие функции-члены). Класс предоставляет определения для динамически создаваемых экземпляров класса (также называются объектами). Классы поддерживают механизмы наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие функциональные возможности базового класса.

Новые классы создаются с помощью объявлений класса. Объявление класса начинается с заголовка, в котором задаются атрибуты и модификаторы класса, имя класса, базовый класс (если есть), а также интерфейсы, реализуемые классом. За заголовком следует тело класса, которое содержит перечень объявлений членов, записанных между разделителями — { и }.

Ниже приведено объявление простого класса Point:

public class Point  
{  
 public int x, y;

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

Экземпляры класса создаются с помощью оператора new, который выделяет память для нового экземпляра, вызывает конструктор для инициализации экземпляра и возвращает ссылку на экземпляр. С помощью следующих операторов создаются два объекта Point, ссылки на которые сохраняются в двух переменных:

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

Память, занимаемая объектом, автоматически освобождается, если объект более не используется. В C# не обязательно (и не допускается) явно освобождать память от объектов.

### Члены

Класс может содержать статические члены или члены экземпляра. Статические члены принадлежат классам. Члены экземпляра принадлежат объектам (экземплярам класса).

В следующей таблице представлен обзор видов членов, которые может содержать класс.

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Константы | Постоянные значения, связанные с классом |
| Поля | Переменные класса |
| Методы | Вычисления и действия, которые могут выполняться классом |
| Свойства | Действия, связанные с чтением и записью именованных свойств класса |
| Индексаторы | Действия, связанные с индексацией экземпляров класса, например массива |
| События | Уведомления, которые могут формироваться классом |
| Операторы | Операторы преобразования и выражений, поддерживаемые классом |
| Конструкторы | Действия, необходимые для инициализации экземпляров класса или самого класса |
| Деструкторы | Действия, выполняемые перед окончательным удалением экземпляров класса |
| Типы | Вложенные типы, объявленные в классе |

### Специальные возможности

Каждому члену класса присваивается собственный уровень доступности, который определяет разделы программы, в которых можно получить доступ к этому члену. Поддерживается пять уровней доступности. Эти уровни представлены в следующей таблице.

|  |  |
| --- | --- |
| **Специальные возможности** | **Значение** |
| public | Доступ не ограничен |
| protected | Доступ ограничен этим классом и унаследованными от него классами |
| internal | Доступ ограничен этой программой |
| protected internal | Доступ ограничен этой программой и классами, унаследованными от этого класса |
| private | Доступ ограничен этим классом |

### Параметры типа

В определении класса может задаваться набор параметров типа. Список имен параметров типа указывается за именем класса в угловых скобках. Параметры типа могут использоваться в теле объявлений класса для определения членов класса. В следующем примере для класса Pair задаются параметры типа TFirst и TSecond:

public class Pair<TFirst,TSecond>  
{  
 public TFirst First;

public TSecond Second;  
}

Тип класса, в объявлении которого принимаются параметры типа, называется универсальным типом класса. Типы структуры, интерфейса и делегата также могут быть универсальными.

При использовании универсального класса для каждого параметра типа необходимо указать аргумент типа:

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

Универсальный тип с предоставленными аргументами типа, например Pair<int,string>, называется сформированным типом.

### Базовые классы

В объявлении класса может задаваться базовый класс, имя которого записывается после имени класса и параметров типа через двоеточие. Класс, для которого не задан базовый класс, считается производным от типа object. В следующем примере для класса Point3D базовым является Point, а для класса Point базовым является object:

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;  
 }  
}

Класс наследует члены своего базового класса. Наследование означает, что класс неявно содержит все члены его базового класса, исключая конструкторы экземпляров и статические конструкторы, а также деструкторы базового класса. Производный класс может добавлять новые члены к своим унаследованным членам, но он не может удалить определение унаследованного члена. В предыдущем примере класс Point3D наследует поля x и y класса Point. Это означает, что каждый экземпляр класса Point3D содержит три поля: x, y и z.

Существует неявное преобразование из типа класса к любому из его базовых типов класса. Таким образом, переменная типа класса может ссылаться на экземпляр этого класса или любого производного от него класса. Например, если сделаны предыдущие объявления классов, переменная типа Point может ссылаться как на экземпляр класса Point, так и на экземпляр класса Point3D:

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

### Поля

Поле представляет собой переменную, связанную с классом или экземпляром класса.

Поле, объявленное с использованием модификатора static, определяет статическое поле. Статическое поле определяет только одно место хранения. Вне зависимости от количества создаваемых экземпляров класса всегда существует только одна копия статического поля.

Поле, объявленное без использования модификатора static, определяет поле экземпляра. Каждый экземпляр класса содержит отдельную копию всех полей экземпляра класса.

В следующем примере каждый экземпляр класса Color содержит отдельную копию полей экземпляра r, g и b. Однако существует только одна копия статических полей Black, White, Red, Green и Blue:

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;  
 }  
}

В предыдущем примере показано объявление полей только для чтения с помощью модификатора readonly. Присваивание значения полю с модификатором readonly может выполняться только при объявлении поля или в конструкторе этого класса.

### Методы

Метод — это член, реализующий вычисление или действие, которое может быть выполнено объектом или классом. Доступ к статическим методам осуществляется через классы. Доступ к методам экземпляра осуществляется через экземпляры класса.

Метод содержит (возможно, пустой) список параметров, которые представляют собой значения или ссылки на переменные, передаваемые в метод, а также тип возвращаемого значения, который задает тип значения, вычисляемого и возвращаемого методом. Если метод не возвращает значение, ему присваивается тип возвращаемого значения void.

Методы, как и типы, могут содержать набор параметров типа, для которых при вызове метода должны указываться аргументы типа. В отличие от типов, аргументы типа зачастую могут выводиться из аргументов вызова метода и не требуют явного определения.

Сигнатура метода должна быть уникальной в пределах класса, в котором он объявляется. Сигнатура метода состоит из имени метода, числа параметров типа, а также числа, модификаторов и типов его параметров. Сигнатура метода не содержит тип возвращаемого значения.

#### Параметры

Параметры используются для передачи в методы значений или ссылок на переменные. Параметры метода получают фактические значения от аргументов, которые указываются при вызове метода. Поддерживается четыре вида параметров: параметры значений, параметры ссылок, выходные параметры и массивы-параметры.

Параметр значения используется для передачи входного параметра. Параметр значения соответствует локальной переменной, начальное значение которой получается из аргумента, передаваемого для этого параметра. Изменение параметра значения не влияет на аргумент, передаваемый для этого параметра.

Параметры значения могут быть необязательными, указывающими значение по умолчанию, поэтому соответствующие аргументы могут быть опущены.

Параметр ссылки используется для передачи как входных, так и выходных параметров. Аргумент, передаваемый параметру ссылки, должен являться переменной. Во время выполнения метода параметр ссылки представляет то же место хранения, что и переменная аргумента. Параметр ссылки объявляется с помощью модификатора ref. В следующем примере показано использование параметров с модификатором ref.

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"  
 }  
}

Выходной параметр используется для передачи выходных параметров. Выходной параметр аналогичен параметру ссылки, однако он не учитывает начальное значение аргумента, предоставляемого вызывающим объектом. Выходной параметр объявляется с помощью модификатора out. В следующем примере показано использование параметров с модификатором out.

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"  
 }  
}

Массив-параметр используется для передачи методу переменного числа аргументов. Массив-параметр объявляется с помощью модификатора params. В качестве массива-параметра может использоваться только последний параметр метода. Массив-параметр должен являться одномерным массивом. Хорошими примерами использования массивов-параметров являются методы Write и WriteLine класса System.Console. Эти методы объявляются следующим образом.

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

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

...  
}

В таких методах массив параметров используется как обычный параметр, имеющий тип массива. Однако при вызове метода с массивом параметров можно передать как один аргумент типа массива параметров, так и любое число аргументов с типом элементов массива параметров. В последнем случае экземпляр массива автоматически создается и инициализируется с использованием заданных аргументов. Пример

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

равнозначен следующей записи:

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);

#### Тело метода и локальные переменные

В теле метода определяются операторы, выполняемые при его вызове.

В теле метода могут быть объявлены переменные, относящиеся к вызову метода. Такие переменные называются локальными переменными. В объявлении локальной переменной задается имя типа, имя переменной и при необходимости начальное значение. В следующем примере объявляются локальная переменная i с нулевым начальным значением и локальная переменная j без начального значения.

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;  
 }  
 }  
}

В C# значение локальной переменной можно использовать только после явного присваивания ей значения. Например, если переменной i (см. выше) не присвоено начальное значение, при последующем использовании i возникнет ошибка компиляции, поскольку в момент использования переменной i ей явно не присвоено значение.

В методе можно использовать операторы return для передачи управления вызвавшему его объекту. В методе, возвращающем значение void, операторы return не могут задавать выражение. В методе, возвращающем отличное от void значение, операторы return должны включать выражение, вычисляющее возвращаемое значение.

#### Статические методы и методы экземпляра

Метод, объявленный с использованием модификатора static, называется статическим методом. Статический метод не выполняет операций с конкретным экземпляром и может напрямую обращаться только к статическим членам.

Метод, объявленный без использования модификатора static, называется методом экземпляра. Метод экземпляра выполняет операции только с конкретным экземпляром и может обращаться как к статическим членам, так и к членам экземпляра. Явное обращение к экземпляру, для которого вызывается метод экземпляра, выполняется с помощью ключевого слова this. При ссылке на this в статическом методе возникает ошибка.

В следующем примере класс Entity содержит как статические члены, так и члены экземпляра.

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;  
 }  
}

Каждый экземпляр класса Entity содержит серийный номер (и, возможно, другие данные, не показанные в примере). Конструктор класса Entity (аналогичен методу экземпляра) инициализирует новый экземпляр со следующим доступным серийным номером. Поскольку конструктор представляет собой член экземпляра, допускается обращение как к полю экземпляра serialNo, так и к статическому полю nextSerialNo.

Статические методы GetNextSerialNo и SetNextSerialNo могут обращаться к статическому полю nextSerialNo, однако при непосредственном обращении этих методов к полю экземпляра serialNo возникнет ошибка.

В следующем примере показано использование класса Entity.

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"  
 }  
}

Обратите внимание, что статические методы SetNextSerialNo и GetNextSerialNo вызываются для класса, а метод экземпляра GetSerialNo — для экземпляра класса.

#### Виртуальные, переопределяющие и абстрактные методы

Если объявление метода экземпляра содержит модификатор virtual, метод является виртуальным методом. Если модификатор virtual отсутствует, метод называется невиртуальным методом.

При вызове виртуального метода тип времени выполнения экземпляра, для которого осуществляется вызов, определяет фактическую реализацию вызываемого метода. При вызове невиртуального метода определяющим фактором является тип времени компиляции экземпляра.

Виртуальный метод может быть переопределен в производном классе. Если объявление метода экземпляра содержит модификатор override, метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Объявление виртуального метода представляет новый метод. Объявление переопределяющего метода уточняет существующий виртуальный метод, предоставляя его новую реализацию.

Абстрактным называется виртуальный метод без реализации. Объявление абстрактного метода осуществляется с использованием модификатора abstract и допускается только в классе, объявленном как abstract. В каждом неабстрактном производном классе необходимо переопределять абстрактный метод.

В следующем примере объявляется абстрактный класс Expression, представляющий узел дерева выражений, а также три производных класса: Constant, VariableReference и Operation, которые реализуют узлы дерева выражений для констант, ссылок на переменные и арифметических операций. (Эти классы похожи на типы дерева выражений, представленные в разделе §4.6. Однако их не следует путать.)

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");  
 }  
}

Четыре приведенных выше класса могут использоваться для моделирования арифметических выражений. Например, с помощью экземпляров этих классов выражение x + 3 можно представить следующим образом.

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

Метод Evaluate экземпляра Expression вызывается для вычисления заданного выражения и возвращает значение типа double. Метод принимает в качестве аргумента параметр Hashtable, содержащий имена переменных (в качестве ключей записей) и значения (в качестве значений записей). Метод Evaluate представляет собой виртуальный абстрактный метод. Это означает, что в производных от него неабстрактных классах необходимо переопределить этот метод и предоставить его фактическую реализацию.

Реализация Constant метода Evaluate лишь возвращает хранящееся значение константы. Реализация VariableReference осуществляет поиск имени переменной в хэш-таблице и возвращает значение результата. Реализация Operation сначала выполняет вычисление левого и правого операндов (посредством рекурсивного вызова соответствующих методов Evaluate), а затем выполняет заданную арифметическую операцию.

В следующей программе классы Expression используются для вычисления выражения x \* (y + 2) с различными значениями x и y.

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"  
 }  
}

#### Перегрузка метода

Перегрузка метода позволяет использовать в одном классе несколько методов с одинаковыми именами и различными сигнатурами. При компиляции вызова перегруженного метода компилятор использует разрешение перегрузки для определения конкретного вызываемого метода. С помощью разрешения перегрузки определяется метод, наиболее подходящий для заданных аргументов, или, если такой метод не найден, возвращается сообщение об ошибке. В следующем примере показано действие разрешения перегрузки. В комментариях к каждому вызову метода Main указывается, какой метод фактически вызывается.

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) }  
}

Как показано в примере, конкретный метод всегда можно выбрать посредством явного приведения аргументов к соответствующим типам параметров или явного предоставления аргументов типа.

### Другие члены-функции

Члены класса, содержащие исполняемый код, в совокупности называются члены-функции. В предыдущем разделе описаны методы, представляющие собой простые члены-функции. В этом разделе описываются другие типы членов-функций, поддерживаемые в C#: конструкторы, свойства, индексаторы, события, операторы и деструкторы.

В следующей таблице описывается универсальный класс List<T>, который реализует расширяемый список объектов. Класс содержит несколько примеров наиболее распространенных типов членов-функций.

|  |  |
| --- | --- |
| public class List<T> { | |
| const int defaultCapacity = 4; | Константа |
| T[] items;  int count; | Поля |
| public List(int capacity = defaultCapacity) {  items = new T[capacity];  } | Конструкторы |
| public int Count {  get { return count; }  }  public int Capacity {  get {  return items.Length;  }  set {  if (value < count) value = count;  if (value != items.Length) {  T[] newItems = new T[value];  Array.Copy(items, 0, newItems, 0, count);  items = newItems;  }  }  } | Свойства |

|  |  |
| --- | --- |
| public T this[int index] {  get {  return items[index];  }  set {  items[index] = value;  OnChanged();  }  } | Индексатор |
| 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<T>);  }  static bool Equals(List<T> a, List<T> b) {  if (a == null) return b == null;  if (b == null || a.count != b.count) return false;  for (int i = 0; i < a.count; i++) {  if (!object.Equals(a.items[i], b.items[i])) {  return false;  }  }  return true;  } | Методы |
| public event EventHandler Changed; | Событие |
| public static bool operator ==(List<T> a, List<T> b) {  return Equals(a, b);  }  public static bool operator !=(List<T> a, List<T> b) {  return !Equals(a, b);  } | Операторы |
| } | |

#### Конструкторы

В C# поддерживаются конструкторы экземпляров и статические конструкторы. Конструктор экземпляра является членом, реализующим действия, необходимые для инициализации экземпляра класса. Статический конструктор представляет собой член, который реализует действие, необходимое для инициализации самого класса при его первой загрузке.

Конструктор объявляется аналогично методу без типа возвращаемого значения и имеет то же имя, что и содержащий его класс. Объявление конструктора, содержащее модификатор static, объявляет статический конструктор. В противном случае объявляется конструктор экземпляра.

Конструкторы экземпляров можно перегружать. Например, в классе List<T> объявляются два конструктора: один без параметров и один, принимающий параметр типа int. Вызов конструктора экземпляра осуществляется с помощью оператора new. В следующем примере выделяются два экземпляра List<string> с использованием каждого из конструкторов класса List.

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

В отличие от других членов, конструкторы экземпляров не могут наследоваться. Класс содержит только те конструкторы экземпляров, которые фактически объявлены в нем. Если в классе не объявлен конструктор экземпляров, автоматически используется пустой конструктор без параметров.

#### Свойства

Свойства представляют собой естественные расширения полей. Как свойства, так и поля являются именованными членами со связанными типами, для обращения к которым используется одинаковый синтаксис. Однако в отличие от полей, свойства не указывают места хранения. Вместо этого свойства содержат методы доступа, определяющие операторы, которые используются при чтении или записи их значений.

Свойство объявляется аналогично полю, однако объявление свойства должно заканчиваться не точкой с запятой, а методами доступа get или set, записанными между разделителями — { и }. Свойство, для которого определены оба метода доступа get и set, называется свойством для чтения и записи. Свойство, для которого определен только метод доступа get, называется свойством только для чтения. Свойство, для которого определен только метод доступа set, называется свойством только для записи.

Метод доступа get соответствует не содержащему параметров методу, возвращаемое значение которого имеет тип свойства. За исключением случаев, когда свойство является конечным объектом операции присваивания, при ссылке на свойство в выражении вызывается метод доступа get для вычисления значения свойства.

Метод доступа set соответствует методу с одним параметром value, не имеющему типа возвращаемого значения. При ссылке на свойство как на целевой объект операции присваивания или как на операнд операторов ++ и -- метод доступа set вызывается с аргументом, который предоставляет новое значение.

В классе List<T> объявляются два свойства: Count и Capacity (только для чтения и только для записи соответственно). В следующем примере показано использование этих свойств.

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

Как и в случае с полями и методами, в C# поддерживаются свойства экземпляров и статические свойства. Свойства, объявленные с использованием модификатора static, называются статическими. Все остальные свойства называются свойствами экземпляров.

Методы доступа свойства могут быть виртуальными. Если объявление свойства содержит модификатор virtual, abstract или override, соответствующий тип применяется и к его методам доступа.

#### Индексаторы

Индексатор — это член, предназначенный для индексации объектов (аналогично массивам). Индексатор объявляется аналогично свойству, однако в качестве имени члена используется this, за которым следует список параметров, записанный между разделителями — [ и ]. Параметры доступны в методах доступа индексатора. Как и свойства, индексаторы могут быть доступны только на чтение, только на запись или на чтение и запись. Методы доступа индексатора могут быть виртуальными.

В классе List объявляется один индексатор для чтения и записи, принимающий параметр int. С его помощью обеспечивается индексация экземпляров класса List с использованием значений типа int. Пример

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();  
}

Индексаторы могут быть перегружены. Это означает, что в классе можно объявить несколько индексаторов, различающихся числом или типом параметров.

#### События

Событие — это член, используемый классом или объектом для предоставления уведомлений. Событие объявляется аналогично полю, однако оно должно иметь тип делегата и его объявление должно содержать ключевое слово event.

Если событие не является абстрактным и не содержит объявления методов доступа, его поведение в классе, в котором оно объявлено, аналогично поведению поля. В поле хранится ссылка на делегат, который представляет обработчики событий, добавленные к событию. Если обработчики событий отсутствуют, поле имеет значение null.

В классе List<T> объявляется член-событие Changed, указывающий на добавление нового элемента в список. Событие Changed вызывается виртуальным методом OnChanged, в котором сначала проверяется, имеет ли событие значение null (т. е. для события отсутствуют обработчики). Понятие вызова события совершенно эквивалентно вызову делегата, представленного событием. Поэтому не существует специальных языковых конструкций для вызова событий.

Реакция клиента на событие реализуется с помощью обработчиков событий. Для добавления обработчиков событий используется оператор «+=, для удаления — оператор -=. В следующем примере к событию Changed класса List<string> присоединяется обработчик событий.

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"  
 }  
}

В расширенных сценариях, в которых требуется управление базовым хранилищем события, в объявлении события можно явно определить методы доступа add и remove, которые во многом аналогичны методу доступа set свойства.

#### Операторы

Оператор — это член, который определяет значение применения конкретного оператора выражения к экземплярам класса. Поддерживается определение операторов трех видов: унарные операторы, двоичные операторы и операторы преобразования. Все операторы должны объявляться с использованием модификаторов public и static.

В классе List<T> объявляются два оператора: operator == и operator !=, что позволяет определить новое значение для выражений, которые применяют эти операторы к экземплярам класса List. В частности, операторы определяют равенство двух экземпляров класса List<T> посредством сравнения всех содержащихся в них объектов с помощью методов Equals. В следующем примере оператор == используется для сравнения двух экземпляров класса List<int>.

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"  
 }  
}

Первое выражение Console.WriteLine возвращает True, поскольку два списка содержат одинаковое число объектов с одинаковыми значениями и в одинаковом порядке. Если бы в классе List<T> не был определен оператор operator ==, первое выражение Console.WriteLine возвращало бы False, поскольку a и b ссылаются на различные экземпляры класса List<int>.

#### Деструкторы

Деструктор является членом, реализующим действия, необходимые для уничтожения экземпляра класса. Деструкторы не могут иметь параметров или модификаторов доступа и не могут быть вызваны явно. Деструктор экземпляра вызывается автоматически в процессе сборки мусора.

Сборщик мусора обладает широкими возможностями по определению момента сборки объектов и запуска деструкторов. В частности, выбор времени для вызова деструктора не является определенным. Деструкторы могут выполняться в любом потоке. По этим и другим причинам реализация деструкторов в классе должна осуществляться только при отсутствии других возможных решений.

Наиболее эффективный подход к уничтожению объектов обеспечивается при использовании оператора using.

## Структуры

Как и классы, структуры представляют собой структуры данных, содержащие члены-данные и члены-функции. Однако, в отличие от классов, структуры имеют тип значений и не требуют выделения памяти в куче. Переменная типа структуры непосредственно хранит данные структуры, тогда как переменная типа класса хранит ссылку на динамически выделяемый объект. Типы структуры не поддерживают определяемое пользователем наследование. Все типы структуры неявно наследуются от типа object.

Структуры особенно удобны для работы с небольшим объемом данных, имеющих семантику значения. Примерами структур являются комплексные числа, точки в системе координат или словарные пары «ключ-значение». Применение структур вместо классов при работе с небольшими структурами данных позволяет добиться значительной экономии выделяемой памяти и повышения производительности приложения. Например, в следующей программе создается и инициализируется массив из 100 точек. Если в качестве класса реализуется Point, создается 101 экземпляр объектов — один для массива и по одному для каждого из 100 его элементов.

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);  
 }  
}

Также можно реализовать Point как структуру.

struct Point  
{  
 public int x, y;

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

В этом случае создается только один экземпляр объекта для массива. Экземпляры Point хранятся встроенными в массив.

Конструктор структур вызывается с помощью оператора new, однако при этом не предполагается выделение памяти. Вместо динамического выделения объектов с возвращением ссылок на них конструктор структуры возвращает значение структуры (обычно во временной ячейке стека), которое затем при необходимости копируется.

Две переменные типа класса могут ссылаться на один объект. Это позволяет изменять объект, на который ссылается одна переменная, выполняя соответствующие операции с другой. Каждая переменная типа структуры содержит собственную копию данных. В связи с этим операции с одной переменной не влияют на другую. Например, выходные данные следующего фрагмента кода зависят от того, является ли Point классом или структурой.

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

Если Point является классом, возвращается значение 20, поскольку a и b ссылаются на один объект. Если Point является структурой, возвращается значение 10, поскольку при присваивании a экземпляру b создается копия значения, на которую не влияет последующее присваивание a.x.

В предыдущем примере показаны два ограничения, накладываемые на структуры. Во-первых, копирование всей структуры обычно менее эффективно, чем копирование ссылки на объект, из-за чего операции присваивания и передачи параметров значения для структур потребляют больше ресурсов, чем аналогичные операции для ссылочных типов. Во-вторых, не поддерживается создание ссылок на структуры (за исключением параметров ref и out), что в некоторых случаях не позволяет использовать структуры.

## Массивы

Массив представляет собой структуру данных, содержащую определенное число переменных, доступ к которым осуществляется с помощью вычисляемых индексов. Все переменные, содержащиеся в массиве, которые также называются элементами массива, имеют одинаковый тип, который называется типом элементов массива.

Типы массивов являются ссылочными типами, поэтому при объявлении переменной типа массива выделяется память для ссылки на экземпляр массива. Фактические экземпляры массива создаются динамически во время выполнения с помощью оператора new. Оператор new определяет длину создаваемого экземпляра массива, которая остается неизменной в течение всего времени существования экземпляра. Элементам массива присваиваются индексы в диапазоне от 0 до Length - 1. Оператор new автоматически инициализирует элементы массива с использованием значений по умолчанию (нули для всех числовых типов или null для всех ссылочных типов).

В следующем примере создается и инициализируется массив элементов типа int, после чего выводится содержимое созданного массива.

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]);  
 }  
 }  
}

В этом примере создается и используется одномерный массив. В C# также поддерживаются многомерные массивы. Число измерений также называется рангом типа массива и определяется как сумма единицы и числа запятых, указанных в квадратных скобках типа массива. В следующем примере выделяется память для одно-, двух- и трехмерного массивов.

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

Массив a1 содержит 10 элементов, массив a2 — 50 (10 × 5) элементов, а массив a3 — 100 (10 × 5 × 2) элементов.

Элемент массива может иметь любой тип, в том числе тип массива. Массив, содержащий элементы типа массива (jagged array, или массив массивов), иногда называется неравномерным массивом, поскольку его элементы могут иметь различную длину. В следующем примере выделяется память для массива массивов типа int:

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

В первой строке создается массив, состоящий из трех элементов типа int[], каждый из которых имеет начальное значение null. В последующих строках инициализируются три элемента со ссылками на отдельные экземпляры массивов различной длины.

Оператор new позволяет задать начальные значения элементов массива с помощью инициализатора массива, который представляет собой список выражений, записанных между разделителями { и }. В следующем примере выделяется и инициализируется массив int[], содержащий три элемента.

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

Обратите внимание, что длина массива определяется на основании числа выражений между фигурными скобками { и }. Поддерживается сокращенная форма объявления локальных переменных и полей, что позволяет не указывать повторно тип массива.

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

Оба предыдущих примера эквивалентны следующему:

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

## Интерфейсы

Интерфейс определяет контракт, который может быть реализован классами и структурами. Интерфейс может содержать методы, свойства, события и индексаторы. Интерфейс не предоставляет реализацию определяемых в нем членов. Вместо этого в нем задаются члены, которые должны предоставляться классами или структурами, реализующими такой интерфейс.

Для интерфейсов допускается множественное наследование. В следующем примере интерфейс IComboBox одновременно наследуется от интерфейсов ITextBox и IListBox.

interface IControl  
{  
 void Paint();  
}

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

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

interface IComboBox: ITextBox, IListBox {}

Классы и структуры способны реализовывать несколько интерфейсов. В следующем примере класс EditBox одновременно реализует интерфейсы IControl и IDataBound.

interface IDataBound  
{  
 void Bind(Binder b);  
}

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

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

Если класс или структура реализует конкретный интерфейс, экземпляры такого класса или такой структуры могут быть неявно преобразованы к типу заданного интерфейса. Пример

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

Если статически неизвестно, реализует ли экземпляр конкретный интерфейс, могут использоваться динамические приведения типов. Например, следующие операторы используют динамические приведения типов для получения реализаций интерфейсов IControl и IDataBound объекта. Поскольку фактическим типом объекта является EditBox, приведения выполняются успешно.

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

В представленном выше классе EditBox метод Paint интерфейса IControl и метод Bind интерфейса IDataBound реализуются с использованием членов с модификатором public. В C# также поддерживается явная реализация членов интерфейса, что позволяет не использовать в классах и структурах члены public. Явная реализация члена интерфейса записывается с использованием полного имени члена интерфейса. Например, класс EditBox может использовать явные реализации членов интерфейса для реализации методов IControl.Paint и IDataBound.Bind следующим образом.

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

void IDataBound.Bind(Binder b) {...}  
}

Обращение к явным членам интерфейса осуществляется исключительно посредством типа интерфейса. Например, реализация метода IControl.Paint, предоставляемая показанным выше классом EditBox, может быть вызвана только с предварительным преобразованием ссылки на EditBox к типу интерфейса IControl.

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

## Перечисления

Перечисляемый тип представляет собой тип значений, содержащий конечное число именованных констант. В следующем примере объявляется и используется перечисляемый тип Color, содержащий три постоянных значения: Red, Green и Blue.

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);  
 }  
}

Каждый перечисляемый тип имеет соответствующий целый тип, называемый базовым типом перечисляемого типа. Если для перечисляемого типа явно не объявлен базовый тип, в качестве базового используется тип int. Формат хранения и диапазон возможных значений перечисляемого типа определяются его базовым типом. Набор значений перечисляемого типа может включать его члены, но не ограничивается только ими. В частности, любое значение базового типа может быть приведено к перечисляемому типу и является отдельным допустимым значением такого типа.

В следующем примере объявляется перечисляемый тип Alignment с базовым типом sbyte:

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

Как показано в приведенном выше примере, объявление члена перечисляемого типа может включать константное выражение, определяющее значение такого члена. Значение константы для каждого члена должно принадлежать диапазону базового типа. Если в объявлении перечисляемого типа значение не задается явно, первому члену типа присваивается нулевое значение, а каждому последующему члену — значение текстуально предшествующего ему члена, увеличенное на единицу.

Значения перечисляемых типов могут быть преобразованы к значениям целых типов (и наоборот) с помощью приведений типов. Пример

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

В качестве значения по умолчанию для любого перечисляемого типа используется целое нулевое значение, преобразованное к перечисляемому типу. Если переменные автоматически инициализируются с использованием значений по умолчанию, такие значения присваиваются переменным перечисляемого типа. Чтобы обеспечить доступность значений по умолчанию перечисляемого типа, литерал 0 неявно преобразуется к любому перечисляемому типу. Таким образом, допускаются выражения следующего вида.

Color c = 0;

## Делегаты

Тип делегата представляет собой ссылки на методы с конкретным списком параметров и типом возвращаемого значения. С помощью делегатов методы обрабатываются как сущности, которым можно передавать параметры и которые можно присваивать переменным. Понятие делегата близко к понятию указателя на функцию, используемому в некоторых других языках. Однако делегаты, в отличие от указателей на функции, представляют собой пример объектно-ориентированного и типобезопасного подхода к программированию.

В следующем примере объявляется и используется тип делегата Function.

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);  
 }  
}

Экземпляр типа делегата Function может ссылаться на любой метод, который принимает аргумент типа double и возвращает значение типа double. Метод Apply применяет заданную функцию Function к элементам массива типа double[] и возвращает массив типа double[], содержащий результаты. В методе Main метод Apply используется для применения трех различных функций к массиву типа double[].

Делегат может ссылаться как на статический метод (например, Square или Math.Sin в предыдущем примере), так и на метод экземпляра (например, m.Multiply в предыдущем примере). Делегат, ссылающийся на метод экземпляра, также ссылается на конкретный объект. При вызове такого метода экземпляра с помощью делегата этот объект становится объектом this.

Делегаты также могут создаваться с помощью анонимных функций, которые представляют собой «встроенные методы», создаваемые в процессе выполнения. Анонимные функции могут видеть локальные переменные окружающих их методов. Таким образом, приведенный выше пример множителя может быть записан проще с использованием класса Multiplier:

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

Интересной и полезной особенностью делегата является то, что для него неважен тип класса или метода, на который он ссылается. Единственным требованием является наличие у метода, на который ссылается делегат, такого же числа параметров и типа возвращаемого значения.

## Атрибуты

Типы, члены и другие сущности C# поддерживают модификаторы, которые управляют определенными аспектами их поведения. Например, доступность метода управляется модификаторами public, protected, internal и private. Благодаря этой возможности в C# пользовательские типы декларативных сведений могут быть вложены в сущности программы и извлекаться во время выполнения. Такие дополнительные декларативные сведения задаются в программе посредством определения и использования атрибутов.

В следующем примере атрибут HelpAttribute присоединяется к сущностям программы и предоставляет ссылки на связанную с ними документацию.

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; }  
 }  
}

Все классы атрибутов наследуются от базового класса System.Attribute, предоставляемого платформой .NET Framework. Чтобы применить атрибут, необходимо указать его имя и любые другие аргументы в квадратных скобках непосредственно перед связанным объявлением. Если имя атрибута заканчивается словом Attribute, при ссылке на него эту часть имени можно опустить. Например, атрибут HelpAttribute можно использовать следующим образом.

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

В этом примере атрибут HelpAttribute присоединяется к классу Widget, а другой атрибут HelpAttribute — к методу Display класса. Общие конструкторы класса атрибута управляют сведениями, которые предоставляются при вложении атрибута в сущность программы. Дополнительные сведения предоставляются посредством ссылки на открытые свойства для чтения и записи класса атрибута (например, ссылка на свойства Topic в предыдущем примере).

В следующем примере показывается извлечение сведений атрибута для заданной сущности программы во время выполнения с помощью отражения.

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"));  
 }  
}

При запросе конкретного атрибута с использованием отражения вызывается конструктор класса атрибута с данными, предоставленными в исходном коде программы, и возвращается результирующий экземпляр атрибута. Если с помощью свойств предоставлены дополнительные сведения, соответствующие значения присваиваются свойствам до возвращения экземпляра атрибута.

# Лексическая структура

## Программы

Программа на C# состоит из одного или более исходных файлов, формально называемых единицами компиляции (§9.1). Исходный файл — это упорядоченная последовательность символов Юникода. Исходные файлы обычно взаимнооднозначно соответствуют файлам файловой системы, но это соответствие не является обязательным. Для максимальной переносимости рекомендуется использовать для файлов в файловой системе кодировку UTF-8.

С концептуальной точки зрения программа компилируется в три этапа:

1. Преобразование: файл преобразуется из конкретного набора символов и схемы кодировки в последовательность символов Юникода.
2. Лексический анализ: поток входных символов Юникода преобразуется в поток лексем.
3. Синтаксический анализ: поток лексем преобразуется в исполняемый код.

## Грамматики

В настоящей спецификации представлен синтаксис языка программирования C#, использующий две грамматики. Лексическая грамматика (§2.2.2) определяет, как объединяются символы Юникода для образования признаков конца строки, пробелов, комментариев, лексем и препроцессорных директив. Синтаксическая грамматика (§2.2.3) определяет, как объединяются лексемы, полученные от лексической грамматики, для образования программ на C#.

### Грамматическая нотация

Лексическая и синтаксическая грамматики представлены с помощью грамматических порождений. Каждое грамматическое порождение определяет нетерминальный символ и возможные подстановки этого нетерминального символа в последовательности нетерминальных или терминальных символов. В грамматических порождениях нетерминальные символы отображаются курсивом, а символы terminal — моноширинным шрифтом.

Первой строкой грамматического порождения является имя определяемого нетерминального символа, за которым следует двоеточие. Каждая последующая строка с отступом содержит возможную подстановку нетерминального символа, данную в виде последовательности нетерминальных или терминальных символов. Например, порождение:

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

определяет оператор while как состоящий из лексемы while, последующей лексемы "(", последующего логического выражения, последующей лексемы ")" и последующего внедренного оператора.

Если существует более одной возможной подстановки нетерминального символа, варианты перечисляются в отдельных строках. Например, порождение:

statement-list:  
statement  
statement-list statement

определяет список операторов как состоящий или из оператора, или из списка операторов с последующим оператором. Иначе говоря, это определение является рекурсивным и указывает, что список операторов состоит из одного или более операторов.

Подстрочный суффикс "opt" используется для указания необязательного символа. Порождение:

block:  
{ statement-listopt }

является краткой записью для:

block:  
{ }  
{ statement-list }

и определяет блок как состоящий из необязательного списка операторов, заключенного в лексемы "{" и "}".

Варианты обычно перечисляются в отдельных строках, но если вариантов много, фраза «one of» («одно из») может предшествовать списку подстановок, заданных в одной строке. Это просто краткая запись вместо перечисления каждого из вариантов отдельной строкой. Например, порождение:

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

является краткой записью для:

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

### Лексика

Лексическая грамматика C# представлена в разделах §2.3, §2.4 и §2.5. Терминальными символами лексической грамматики являются символы набора символов Юникода. Лексическая грамматика определяет, как символы объединяются для образования лексем (§2.4), пробелов (§2.3.3), комментариев (§2.3.2) и препроцессорных директив (§2.5).

Каждый исходный файл программы на C# должен соответствовать порождению ввода лексической грамматики (§2.3).

### Синтаксическая грамматика

Синтаксическая грамматика C# представлена в главах и приложениях, следующих за этой главой. Терминальными символами синтаксической грамматики являются лексемы, определенные лексической грамматикой. Синтаксическая грамматика определяет, как лексемы объединяются для образования программ на C#.

Каждый исходный файл в программе на C# должен соответствовать порождению единицы компиляции синтаксической грамматики (§9.1).

## Лексический анализ

Порождение ввода определяет лексическую структуру исходного файла на C#. Каждый исходный файл в программе на C# должен соответствовать этому порождению лексической грамматики.

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

Лексическую структуру исходного файла на C# составляют пять основных элементов: признаки конца строки (§2.3.1), пробелы (§2.3.3), комментарии (§2.3.2), лексемы (§2.4) и препроцессорные директивы (§2.5). Из этих основных элементов только лексемы являются значимыми в синтаксической грамматике программы на C# (§2.2.3).

Лексическая обработка исходного файла на C# состоит из сведения файла к последовательности лексем, становящейся вводом для синтаксического анализа. Признаки конца строки, пробелы и комментарии могут служить для разделения лексем; препроцессорные директивы могут вызывать пропуск разделов исходного файла, однако в других отношениях эти лексические элементы не влияют на синтаксическую структуру программы на C#.

Если несколько порождений лексической грамматики соответствуют последовательности символов в исходном файле, лексическая обработка всегда создает самый длинный возможный лексический элемент. Например, последовательность символов // обрабатывается как начало однострочного комментария, так как этот лексический элемент длиннее, чем лексема /.

### Знаки завершения строки

Признаки конца строки разделяют на строки символы исходного файла на C#.

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)

Для совместимости со средствами редактирования исходного кода, добавляющими маркеры конца файла, и для возможности просмотра исходного файла как последовательности правильно завершенных строк, к каждому исходному файлу в программе на C# применяются следующие преобразования в таком порядке:

* если последним символом исходного файла является символ CTRL-Z (U+001A), этот символ удаляется;
* символ возврата каретки (U+000D) добавляется к концу исходного файла, если файл не пустой и если последним символом исходного файла не является символ возврата каретки (U+000D), символ перевода строки (U+000A), символ разделителя строк (U+2028) или символ разделителя абзацев (U+2029).

### Комментарии

Поддерживается две формы комментариев: однострочные комментарии и комментарии с разделителями. Однострочные комментарии начинаются с символов // и продолжаются до конца исходной строки. Комментарии с разделителями начинаются с символов /\* и заканчиваются символами \*/. Комментарии с разделителями могут занимать несколько строк.

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 \*

Комментарии не могут быть вложенными. Последовательности символов /\* и \*/ не имеют специального значения внутри комментария //, а последовательности символов // и /\* не имеют специального значения внутри комментария с разделителями.

Комментарии не обрабатываются внутри символьных и строковых литералов.

Пример:

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

содержит комментарий с разделителями.

Пример:

// 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");  
 }  
}

показано несколько однострочных комментариев.

### Пробел

Пробел определяется как любой символ Юникода класса Zs (который включает символ пробела), а также как символ горизонтальной табуляции, символ вертикальной табуляции и символ перевода страницы.

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

## Лексемы

Существует несколько видов лексем: идентификаторы, ключевые слова, литералы, операторы и знаки пунктуации. Пробел и комментарии не являются лексемами, хотя действуют как разделители для лексем.

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

### Управляющие последовательности символов Юникода

Управляющая последовательность символов Юникода представляет собой символ Юникода. Управляющие последовательности символов Юникода обрабатываются в идентификаторах (§2.4.2), символьных литералах (§2.4.4.4) и правильных строковых литералах (§2.4.4.5). Управляющая последовательность символов Юникода не обрабатывается в других местах (например, для образования оператора, знака пунктуации или ключевого слова).

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 hex-digit

Управляющая последовательность Юникода (escape-последовательность Юникода) представляет собой один символ Юникода, образованный шестнадцатеричным числом, следующим за символами "\u" или "\U". Поскольку в C# используется 16-разрядная кодировка элементов кода Юникода в символьных и строковых значениях, символ Юникода в диапазоне от U+10000 до U+10FFFF запрещен в строковом литерале и представляется с помощью суррогатной пары Юникода в строковом литерале. Символы Юникода с элементами кода выше 0x10FFFF не поддерживаются.

Многократные трансляции не выполняются. Например, строковый литерал "\u005Cu005C" эквивалентен "\u005C", а не "\". Значение Юникода \u005C является символом "\".

Пример:

class Class1  
{  
 static void Test(bool \u0066) {  
 char c = '\u0066';  
 if (\u0066)  
 System.Console.WriteLine(c.ToString());  
 }   
}

показано несколько использований \u0066, escape-последовательности для буквы "f". Эта программа эквивалентна следующей:

class Class1  
{  
 static void Test(bool f) {  
 char c = 'f';  
 if (f)  
 System.Console.WriteLine(c.ToString());  
 }   
}

### Идентификаторы

Правила для идентификаторов в этом разделе точно соответствуют правилам, рекомендованным в дополнении 31 к стандарту Юникода, за исключением следующего: знак подчеркивания разрешен в качестве начального символа (что традиционно для языка программирования C), escape-последовательности Юникода разрешены в идентификаторах, а символ "@" разрешен в качестве префикса, чтобы можно было использовать ключевые слова в качестве идентификаторов.

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

Дополнительные сведения об упомянутых выше классах символов Юникода см. в документе Стандарт Юникода, версия 3.0, раздел 4.5.

Примеры допустимых идентификаторов: "identifier1", "\_identifier2" и "@if".

Идентификатор в соответствующей программе должен быть в каноническом формате, определенном формой нормализации C Юникода, как определено в дополнении 15 к стандарту Юникода. Поведение при обнаружении идентификатора не в форме нормализации C определяется реализацией, однако диагностика при этом не требуется.

Префикс "@" позволяет использовать ключевые слова в качестве идентификаторов, что полезно при взаимодействии с другими языками программирования. Символ @ фактически не является частью идентификатора, так что этот идентификатор может отображаться в других языках в виде обычного идентификатора, без префикса. Идентификатор с префиксом @ называется буквальным идентификатором. Использование префикса @ для идентификаторов, не являющихся ключевыми словами, разрешено, но настоятельно не рекомендуется со стилистической точки зрения.

Пример:

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

class Class1  
{  
 static void M() {  
 cl\u0061ss.st\u0061tic(true);  
 }  
}

определяется класс class со статическим методом с именем static, который принимает параметр с именем bool. Обратите внимание, что, поскольку управляющие последовательности Юникода не разрешены в ключевых словах, лексема "cl\u0061ss" является идентификатором, причем тем же идентификатором, что и "@class".

Два идентификатора считаются одинаковыми, если они идентичны после применения следующих преобразований в таком порядке:

* если используется префикс "@", он удаляется;
* каждая последовательность управляющих символов Юникода преобразуется в соответствующий символ Юникода;
* все символы управления форматом удаляются.

Идентификаторы, содержащие два последовательных символа подчеркивания (U+005F), зарезервированы для использования в реализации. Например, реализация может предоставлять расширенные ключевые слова, начинающиеся с двух символов подчеркивания.

### Ключевые слова

Ключевое слово — это подобная идентификатору зарезервированная последовательность символов. Ключевое слово нельзя использовать в качестве идентификатора за исключением его использования с префиксом @.

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

В некоторых положениях грамматики особые идентификаторы имеют особое значение, но не являются ключевыми словами. Такие идентификаторы иногда называются «контекстными ключевыми словами». Например, в объявлении свойства идентификаторы get и set имеют особое значение (§10.7.2). Идентификатор, отличный от get или set, никогда не разрешается в этих положениях, поэтому такое использование не приводит к конфликту с использованием этих слов в качестве идентификаторов. В других случаях, как, например, с идентификатором var в объявлениях с неявно типизированной локальной переменной (§8.5.1), контекстное ключевое слово может конфликтовать с объявленными именами. Тогда объявленное имя получает приоритет над использованием идентификатора в качестве контекстного ключевого слова.

### Литералы

Литерал — это представление значения в исходном коде.

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

#### Логические литералы

Логический литерал может иметь два значения: true и false.

boolean-literal:  
true  
false

Логический литерал имеет тип bool.

#### Целочисленные литералы

Целочисленные литералы используются для записи значений типа int, uint, long или ulong. Целочисленные литералы имеют две возможных формы: десятичную и шестнадцатеричную.

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits integer-type-suffixopt

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 uL ul 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

Тип целочисленного литерала определяется следующим образом:

* если литерал без суффикса, его тип — первый из этих типов, в которых его значение может быть представлено: int, uint, long, ulong;
* если литерал имеет суффикс U или u, он относится к первому из этих типов, в которых его значение может быть представлено как uint или ulong;
* если литерал с суффиксом L или l, его тип — первый из этих типов, в которых его значение может быть представлено: long и ulong;
* если литерал имеет суффикс UL, Ul, uL, ul, LU, Lu, lU или lu, он относится к типу ulong.

Если представленное целочисленным литералом значение находится за пределами диапазона типа ulong, выдается ошибка времени компиляции.

Рекомендуется использовать L вместо l при записи литералов типа long, так как букву l легко спутать с цифрой 1.

Для записи минимально возможных значений int и long в виде десятичных целочисленных литералов существуют следующие два правила:

* если десятичный целочисленный литерал со значением 2147483648 (231) и без суффикса целочисленного типа появляется в качестве лексемы, непосредственно следующей за лексемой оператора унарного минуса (§7.7.2), результатом является константа типа int со значением −2147483648 (−231). Во всех других ситуациях десятичный целочисленный литерал относится к типу uint.
* если десятичный целочисленный литерал со значением 9223372036854775808 (263) и без суффикса целочисленного типа или с суффиксом целочисленного типа L или l появляется в качестве лексемы, непосредственно следующей за лексемой оператора унарного минуса (§7.7.2), результатом является константа типа long со значением −9223372036854775808 (−263). Во всех других ситуациях десятичный целочисленный литерал имеет тип ulong;

#### Действительные литералы

Действительные литералы используются для записи значений типа float, double или decimal.

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt decimal-digits

sign: one of  
+ -

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

Если суффикс действительного типа не указан, типом действительного литерала является double. Иначе суффикс действительного типа определяет тип действительного литерала следующим образом:

* Действительный литерал с суффиксом F или f относится к типу float. Например, каждый из литералов 1f, 1.5f, 1e10f и 123.456F относится к типу float.
* действительный литерал с суффиксом D или d имеет тип double. Например, литералы 1d, 1.5d, 1e10d и 123.456D все имеют тип double;
* действительный литерал с суффиксом M или m имеет тип decimal. Например, литералы 1m, 1.5m, 1e10m и 123.456M все имеют тип decimal; Этот литерал преобразуется в значение типа decimal, принимая его точное значение и при необходимости округляя до ближайшего могущего быть представленным значения с помощью банковского округления (§4.1.7). Любой масштаб, видимый в литерале, сохраняется, если только значение не округляется и не равно нулю (в этом последнем случае знак и масштаб будут равны 0). Следовательно, синтаксический разбор литерала 2.900m создаст десятичное значение со знаком 0, коэффициентом 2900 и масштабом 3.

Если указанный литерал не удается представить в предписанном типе, выдается ошибка времени компиляции.

Значение действительного литерала с типом float или double определяется с помощью режима "округления до ближайшего" по стандарту IEEE.

Обратите внимание, что в действительном литерале всегда требуются десятичные цифры после десятичной точки. Например, 1.3F — это действительный литерал, а 1.F — нет.

#### Символьные литералы

Символьный литерал представляет один символ и обычно состоит из символа в кавычках, например 'a'.

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-digitopt hex-digitopt hex-digitopt

Символ, следующий за обратной косой чертой (\) в символе, должен быть одним из следующих: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Иначе возникает ошибка времени компиляции.

Шестнадцатеричная escape-последовательность представляет собой один символ Юникода со значением, образованным шестнадцатеричным числом, следующим за \x.

Если значение, представленное символьным литералом, больше U+FFFF, вызывается ошибка времени компиляции.

Управляющая последовательность символов Юникода (§2.4.1) в символьном литерале должна быть в диапазоне от U+0000 до U+FFFF.

Простая управляющая последовательность представляет собой кодировку символа Юникода, как показано в следующей таблице.

|  |  |  |
| --- | --- | --- |
| **Escape-последовательность** | **Имя символа** | **Кодировка Юникода** |
| \' | Одинарная кавычка | 0x0027 |
| \" | Двойные кавычки | 0x0022 |
| \\ | Обратная косая черта | 0x005C |
| \0 | Null | 0x0000 |
| \a | Предупреждение | 0x0007 |
| \b | Возврат | 0x0008 |
| \f | Перевод страницы | 0x000C |
| \n | Новая строка | 0x000A |
| \r | Возврат каретки | 0x000D |
| \t | Горизонтальная табуляция | 0x0009 |
| \v | Вертикальная табуляция | 0x000B |

Символьный литерал относится к типу char.

#### Строковые литералы

В C# поддерживается две формы строковых литералов: правильные строковые литералы и буквальные строковые литералы.

Правильный строковый литерал состоит из нуля или более символов, заключенных в двойные кавычки, например "hello", и может включать как простые управляющие последовательности (например \t для символа табуляции), так и шестнадцатеричные escape-последовательности и escape-последовательности Юникода.

Буквальный строковый литерал состоит из символа @ с последующими символом двойных кавычек, нулем или более символов и закрывающим символом двойных кавычек. Простым примером является @"hello". В буквальном строковом литерале символы между разделителями интерпретируются буквально, единственным исключением является управляющая последовательность кавычки. В частности, простые управляющие последовательности, шестнадцатеричные escape-последовательности и escape-последовательности Юникода не обрабатываются в буквальных строковых литералах. Буквальный строковый литерал может занимать несколько строк.

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

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:  
""

Символ, следующий за символом обратной косой черты (\) в символе правильного строкового литерала, должен быть одним из следующих символов: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Иначе возникает ошибка времени компиляции.

Пример:

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\ntwo\r\nthree";  
string j = @"one  
two  
three";

показаны разнообразные строковые литералы. Последний строковый литерал j является буквальным строковым литералом, занимающим несколько строк. Символы между кавычками, включая пробелы, например символы новой строки, сохраняются буквально.

Поскольку в шестнадцатеричной escape-последовательности может быть переменное число шестнадцатеричных цифр, строковый литерал "\x123" содержит один символ с шестнадцатеричным значением 123. Чтобы создать строку, содержащую символ с шестнадцатеричным значением 12 и с последующим символом 3, можно написать "\x00123" или "\x12" + "3".

Строковый литерал имеет тип string.

Результатом каждого строкового литерала не обязательно является новый экземпляр строки. Если два или более строковых литерала, эквивалентных согласно оператору равенства строк (§7.10.7), появляются в одной программе, эти строковые литералы ссылаются на один и тот же экземпляр строки. Например, программа

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

выведет True, так как эти два литерала ссылаются на один и тот же экземпляр строки.

#### Литерал null

null-literal:  
null

Литерал NULL может быть неявно преобразован в ссылочный тип или обнуляемый тип.

### Операторы и знаки пунктуации

Существует несколько видов операторов и знаков пунктуации. Операторы используются в выражениях для описания операций, включающих один или более операндов. Например, в выражении a + b используется оператор + для сложения двух операндов a и b. Знаки пунктуации служат для группирования и разделения.

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

right-shift:  
>|>

right-shift-assignment:  
>|>=

Вертикальная черта в порождениях сдвиг вправо и присваивание сдвига вправо указывает, что, в отличие от других порождений в синтаксической грамматике, никакие символы (даже пробелы) не допустимы между этими лексемами. Эти порождения обрабатываются особо, чтобы сделать возможной правильную обработку списков параметров типа (§10.1.3).

## Препроцессорные директивы

Препроцессорные директивы дают возможность условно пропускать разделы исходных файлов, сообщать об условиях ошибок и предупреждений и устанавливать отдельные области исходного кода. Термин «препроцессорные директивы» используется только для сохранения единообразия с языками программирования C и C++. В C# нет отдельного шага предварительной обработки; препроцессорные директивы обрабатываются как часть этапа лексического анализа.

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

Доступны следующие препроцессорные директивы:

* #define и #undef, используемые, соответственно, для определения и отмены определения символов условной компиляции (§2.5.3);
* #if, #elif, #else и #endif, используемые для условного пропуска разделов исходного кода (§2.5.4);
* #line, используемая для управления номерами строк, выдаваемыми для ошибок и предупреждений (§2.5.7);
* #error и #warning, используемые для выдачи, соответственно, ошибок и предупреждений (§2.5.5);
* #region и #endregion, используемые для явного выделения разделов исходного кода (§2.5.6);
* #pragma, используемая для задания компилятору необязательных сведений о контексте (§2.5.8).

Препроцессорная директива всегда занимает отдельную строку исходного кода и всегда начинается с символа # и имени препроцессорной директивы. Пробел может стоять перед символом # и между символом # и именем директивы.

Строка исходного кода, содержащая директиву #define, #undef, #if, #elif, #else, #endif, #line или #endregion, может оканчиваться однострочным комментарием. Комментарии с разделителями (вид комментариев с /\* \*/) не разрешены в исходных строках, содержащих препроцессорные директивы.

Препроцессорные директивы не являются лексемами и не являются частью синтаксической грамматики C#. Тем не менее, препроцессорные директивы можно использовать для включения или исключения последовательностей лексем и таким образом влиять на значение программы C#. Например, скомпилированная программа:

#define A  
#undef B

class C  
{  
#if A  
 void F() {}  
#else  
 void G() {}  
#endif

#if B  
 void H() {}  
#else  
 void I() {}  
#endif  
}

приведет к той же последовательности лексем, что и программа:

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

Таким образом, несмотря на то, что лексически эти программы различаются, синтаксически они идентичны.

### Символы условной компиляции

Функциональность условной компиляции, предоставляемая директивами #if, #elif, #else и #endif, управляется посредством препроцессорных выражений (§2.5.2) и символов условной компиляции.

conditional-symbol:  
Any identifier-or-keyword except true or false

У символа условной компиляции есть два возможных состояния: определенное и неопределенное. В начале лексической обработки исходного файла символ условной компиляции не определен, если только он явным образом не определен внешним механизмом (таким как параметр командной строки компилятора). При обработке директивы #define символ условной компиляции, упомянутый в этой директиве, становится определенным в этом исходном файле. Этот символ остается определенным, пока не будет обработана директива #undef для того же самого символа, или пока не будет достигнут конец исходного файла. Это означает, что директивы #define и #undef в одном исходном файле не влияют на другие исходные файлы той же программы.

При ссылке в препроцессорном выражении определенный символ условной компиляции имеет логическое значение true, а неопределенный — логическое значение false. Символы условной компиляции не обязательно должны быть явно объявлены, чтобы на них можно было ссылаться в препроцессорных выражениях. Необъявленные символы просто являются неопределенными и имеют, таким образом, значение false.

Пространство имен для символов условной компиляции отличается и является отдельным от всех других именованных сущностей в программе на C#. На символы условной компиляции можно ссылаться только в директивах #define и #undef и в препроцессорных выражениях.

### Препроцессорные выражения

Препроцессорные выражения могут быть в директивах #if и #elif. В препроцессорных выражениях разрешены операторы !, ==, !=, && и ||, для группирования можно использовать скобки.

pp-expression:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

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

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

При ссылке в препроцессорном выражении определенный символ условной компиляции имеет логическое значение true, а неопределенный — логическое значение false.

Вычисление препроцессорного выражения всегда дает логическое значение. Правила вычисления для препроцессорного выражения такие же, как для константного выражения (§7.19), с тем исключением, что единственные определенные пользователем сущности, на которые можно ссылаться, — это символы условной компиляции.

### Директивы объявлений

Директивы объявлений используются для определения или отмены определения символов условной компиляции.

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

Обработка директивы #define делает данный символ условной компиляции определенным, начиная с исходной строки, следующей за директивой. Подобным же образом обработка директивы #undef делает данный символ условной компиляции неопределенным, начиная с исходной строки, следующей за директивой.

Все директивы #define и #undef в исходном файле должны стоять до первой лексемы (§2.4) в исходном файле, иначе вызывается ошибка времени компиляции. Интуитивно понятно, что директивы #define и #undef должны предшествовать любому "фактическому коду" в исходном файле.

Пример:

#define Enterprise

#if Professional || Enterprise  
 #define Advanced  
#endif

namespace Megacorp.Data  
{  
 #if Advanced  
 class PivotTable {...}  
 #endif  
}

правильно, так как директивы#define предшествуют первой лексеме (ключевое слово namespace) в исходном файле.

Следующий пример приводит к ошибке времени компиляции, так как директива #define следует за фактическим кодом:

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

Директива #define может определять уже определенный символ условной компиляции, без какого-либо вмешательства директивы #undef для этого символа. В следующем примере символ условной компиляции A определен, а затем определен снова.

#define A  
#define A

Директива #undef может "отменить определение" не определенного символа условной компиляции. В следующем примере определен символ условной компиляции A, а затем дважды отменено его определение; хотя вторая директива #undef не действует, она все же допустима.

#define A  
#undef A  
#undef A

### Директивы условной компиляции

Директивы условной компиляции используются для условного включения или исключения частей исходного файла.

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 #

В соответствии с синтаксисом, директивы условной компиляции должны быть записаны как наборы, состоящие из (по порядку) директивы #if, нуля или более директив #elif, нуля или одной директивы #else и директивы #endif. Между этими директивами находятся условные разделы исходного кода. Каждый раздел управляется непосредственно предшествующей директивой. Условный раздел может содержать вложенные директивы условной компиляции при условии, что они образуют полные наборы.

ПП условное выражение выделяет не более одного из содержащихся условных разделов для обычной лексической обработки:

* ПП выражения директив #if и #elif вычисляются по порядку, пока одна из них не выдаст значение true. Если выражение выдает true, выбирается условный раздел соответствующей директивы;
* если все ПП выражения дают false и если имеется директива #else, выбирается условный раздел директивы #else;
* иначе условный раздел не выбирается.

Выбранный условный раздел, если он есть, обрабатывается как обычный раздел ввода: в исходном коде, содержащемся в этом разделе, должна соблюдаться лексическая грамматика; лексемы создаются из исходного кода в разделе, а препроцессорные директивы в разделе имеют предписанные действия.

Оставшиеся условные разделы, если они есть, обрабатываются как пропущенные разделы: за исключением препроцессорных директив, исходный код в разделе не обязан соблюдать лексическую грамматику; в этом разделе не создаются лексемы из исходного кода, а препроцессорные директивы должны быть лексически правильными, но в других отношениях они не обрабатываются. Внутри условного раздела, который обрабатывается как пропущенный раздел, любые вложенные условные разделы (содержащиеся во вложенных конструкциях #if...#endif и #region...#endregion) также обрабатываются как пропущенные разделы.

В следующем примере показано, как могут вкладываться директивы условной компиляции:

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

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

За исключением препроцессорных директив, пропущенный исходный код не является предметом лексического анализа. Например, следующее допустимо, несмотря на незавершенный комментарий в разделе #else:

#define Debug // Debugging on

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

При этом следует учитывать, что препроцессорные директивы должны быть лексически правильными даже в пропущенных разделах исходного кода.

Препроцессорные директивы не обрабатываются, если они находятся внутри элементов ввода в несколько строк. Например, программа

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

имеет результатом:

hello,  
#if Debug  
 world  
#else  
 Nebraska  
#endif

В особенных случаях обрабатываемый набор препроцессорных директив может зависеть от вычисления ПП-выражения. Пример:

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

в любом случае создается тот же самый поток лексем (class Q { }), независимо от того, определен X или нет. Если X определен, обрабатываются только директивы #if и #endif из-за многострочного комментария. Если X не определен, в набор директив входят три директивы (#if, #else и #endif).

### Директивы диагностики

Директивы диагностики используются для явного создания сообщений об ошибках и предупреждениях, передаваемых тем же способом, что и другие ошибки и предупреждения времени компиляции.

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

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

Пример:

#warning Code review needed before check-in

#if Debug && Retail  
 #error A build can't be both debug and retail  
#endif

class Test {...}

в любом случае выдается предупреждение ("Code review needed before check-in") и создается ошибка времени компиляции ("A build can’t be both debug and retail"), если определены оба условных символа Debug и Retail. Обратите внимание, что ПП-сообщение может содержать произвольный текст; в частности, оно не должно содержать правильно сформированные лексемы; об этом свидетельствует использование одиночной кавычки в слове can’t.

### Директивы областей

Директивы областей используются для явного выделения областей исходного кода.

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

В области не вложены семантические значения; области используют программисты и автоматические средства, чтобы пометить раздел исходного кода. Сообщение, указанное в директиве #region или #endregion, также не имеет семантического значения, оно служит только для идентификации области. В соответствующих директивах #region и #endregion могут содержаться разные ПП\_сообщения.

Лексическая обработка области

#region  
...  
#endregion

точно соответствует лексической обработке директивы условной компиляции следующей формы:

#if true  
...  
#endif

### Директивы строк

Директивы строк можно использовать для изменения номеров строк и имен исходных файлов, которые сообщаются в выходных данных компилятора, например в предупреждениях и ошибках, или используются в атрибутах сведений о вызывающем коде (§17.4.4).

Директивы строк обычно используются в средствах метапрограммирования, создающих исходный код на C# из другого текстового ввода.

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 "

При отсутствии директив #line компилятор сообщает в своем выводе настоящие номера строк и имена исходных файлов. При обработке директивы #line, содержащей индикатор строки, не являющийся default, компилятор обрабатывает строку после этой директивы как строку с указанным номером (и именем файла, если оно указано).

Директива #line default отменяет действие всех предшествующих директив #line. Компилятор сообщает настоящие сведения о строках для последующих строк, как если бы директивы #line не обрабатывались.

Директива #line hidden не влияет на файл и номера строк, передаваемые в сообщениях об ошибках, но влияет на отладку на исходном уровне. При отладке все строки между директивой #line hidden и последующей директивой #line (отличной от директивы #line hidden) не имеют сведений о номерах строк. При пошаговом выполнении кода в отладчике эти строки полностью пропускаются.

Обратите внимание, что имя файла отличается от правильных строковых литералов тем, что escape-символы не обрабатываются; символ "\" просто означает обычный символ обратной косой черты в имени файла.

### Директивы pragma

Препроцессорная директива #pragma используется для указания компилятору необязательных сведений о контексте. Сведения, предоставленные в директиве #pragma, ни в каком случае не изменяют семантику программы.

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

pragma-body:  
pragma-warning-body

C# предоставляет директивы #pragma для управления предупреждениями компилятора. В будущие версии языка могут быть включены дополнительные директивы #pragma. Чтобы обеспечить взаимодействие с другими компиляторами C#, компилятор C# корпорации Майкрософт не выдает ошибки компиляции для неизвестных директив #pragma; тем не менее, такие директивы создают предупреждения.

#### Директива pragma warning

Директива #pragma warning используется для отключения или восстановления всех или отдельного набора предупреждающих сообщений во время компиляции последующего текста программы.

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

Директива #pragma warning, в которой опущен список предупреждений, влияет на все предупреждения. Директива #pragma warning, в которую включен список предупреждений, влияет только на предупреждения, указанные в этом списке.

Директива #pragma warning disable отключает все предупреждения или заданный набор предупреждений.

Директива #pragma warning restore восстанавливает все предупреждения или заданный набор до состояния, бывшего в начале блока компиляции. Обратите внимание, что если отдельное предупреждение было отключено извне, директива #pragma warning restore (для всех или для конкретного предупреждения) не включит это предупреждение.

В следующем примере показано использование директивы #pragma warning для временного отключения предупреждений, выдаваемых при ссылке на устаревшие члены, с помощью номера предупреждения компилятора Microsoft C#.

using System;

class Program  
{  
 [Obsolete]  
 static void Foo() {}

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

# Основные понятия

## Запуск приложения

Сборка с точкой входа называется приложением. При выполнении приложения создается новый домен приложения. Несколько различных экземпляров приложения могут существовать на одном компьютере одновременно, при этом у каждого будет свой домен приложения.

Домен приложения обеспечивает изоляцию приложения путем функционирования в качестве контейнера состояния приложения. Домен приложения функционирует в качестве контейнера и границы типов, заданных в приложении, и используемых библиотек классов. Типы, загружаемые в один домен приложения, отличаются от аналогичных типов, загруженных в другой домен приложения, а экземпляры объектов не используются совместно непосредственным образом между доменами приложений. Например, каждый домен приложения имеет свою копию статических переменных для данных типов, а статический конструктор для типа выполняется не более одного раза на домен приложения. Можно обеспечивать характерные реализации политики или механизмы для создания или удаления доменов приложений.

Запуск приложений происходит, когда среда выполнения вызывает назначенный метод, известный как точка входа приложения. Этот метод точки входа всегда носит имя Main и может иметь одну из следующих сигнатур:

static void Main() {...}

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

static int Main() {...}

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

Как указано выше, точка входа может дополнительно возвратить значение int. Данное возвращаемое значение используется в завершении приложения (§3.2).

Точка входа может дополнительно иметь один формальный параметр. Параметр может иметь любое имя, но тип параметра должен быть string[]. При наличии формального параметра среда выполнения создает и передает аргумент string[], содержащий аргументы командной строки, заданные при запуске приложения. Аргумент string[] никогда не принимает значение NULL, но может иметь нулевую длину, если не были заданы аргументы командной строки.

Так как C# поддерживает перегрузку методов, класс или структура могут содержать несколько определений какого-либо метода при условии того, что их сигнатуры отличаются. Однако в рамках одной программы класс или структуры не могут содержать более одного метода Main, определение которого позволяет использовать этот метод в качестве точки входа приложения. Другие перегруженные версии Main разрешены с условием наличия более одного параметра или с условием наличия только параметра, тип которого не является типом string[].

Приложение может содержать несколько классов и структур. При этом более одного класса или более одной структуры могут содержать метод Main, определение которого позволяет использовать его в качестве точки входа приложения. В таких случаях для выбора одного из методов Main в качестве точки входа требуется использовать внешний механизм (например, компилятор командной строки).

В C# каждый метод должен быть задан в качестве члена класса или структуры. Как правило, объявленная доступность (§3.5.1) метода определяется модификаторами доступа (§10.3.5), указанными в объявлении, и, аналогичным образом, объявленная доступность типа определяется модификаторами доступа, указанными в объявлении. Чтобы заданный метод заданного типа можно было вызвать, и тип, и член должны быть доступны. Однако точка входа приложения является особым случаем. В частности, среда выполнения может вызвать точку доступа приложения независимо от объявленной доступности в объявлениях вмещающих типов.

Метод точки входа приложения может не находиться в объявлении универсального класса.

Во всех других отношениях методы точки входа функционируют аналогично другим методам.

## Завершение приложения

При завершении приложения управление возвращается среде выполнения.

Если возвращаемое значение метода точки входа приложения имеет тип int, то оно служит кодом состояния завершения приложения. Целью данного кода является передача среде выполнения сведений о том, успешно ли прошло выполнение, или имел место сбой.

Если возвращаемое значение метода точки входа имеет тип void, то достижение правой фигурной скобки (}), завершающей данный метод, или выполнение оператора return без выражения приведет к коду статуса завершения со значением 0.

Перед завершением приложения для всех его объектов, которые еще не были собраны сборщиком мусора, вызываются деструкторы, если такая очистка не была отключена (например, путем вызова метода библиотеки GC.SuppressFinalize).

## Объявления

Объявления в программе на C# определяют составные элементы программы. Программы на C# организованы посредством использования пространств имен (§9), которые могут содержать объявления типов и объявления вложенных пространств имен. Объявления типов (§9.6) используются для задания классов (§10), структур (§10.14), интерфейсов (§13), перечислений (§14) и делегатов (§15). Типы членов, разрешенных в объявлении типов, зависят от формы объявления типа. Например, объявления классов могут содержать объявления для констант (§10.4), полей (§10.5), методов (§10.6), свойств (§10.7), событий (§10.8), индексаторов (§10.9), операторов (§10.10), конструкторов экземпляров (§10.11), статических конструкторов (§10.12), деструкторов (§10.13) и вложенных типов (§10.3.8).

Объявление определяет имя в области объявления, которому принадлежит объявление. За исключением перегруженных членов (§3.6), наличие двух или более объявлений, представляющих члены с одним именем в области объявления, является ошибкой времени компиляции. Недопустимо, чтобы область объявления содержала различные типы членов с одинаковыми именами. Например, область объявления не может содержать поля и метода с одинаковыми именами.

Существует несколько типов областей объявлений, которые представлены далее.

* В рамках всех исходных файлов программы объявления членов пространства имен без вмещающего объявления пространства имен являются членами единой комбинированной области объявления, которая называется глобальной областью объявления.
* В рамках всех исходных файлов программы объявления членов пространства имен в объявлении пространства имен с аналогичным полным именем пространства имен являются членами одной комбинированной области объявления.
* Каждый класс, структура или интерфейс объявления создает новую область объявления. Имена размещаются в данной области объявления посредством объявлений членов классов, объявлений членов структур, объявлений членов интерфейсов или параметров типов. Исключая объявления перегруженных конструкторов экземпляров и объявления статических конструкторов, классы или структуры не могут содержать объявления членов с именем, совпадающим с именем класса или структуры. Класс, структура или интерфейс допускает объявление перегруженных методов и индексаторов. Более того, класс или структура допускают объявление перегруженных конструкторов экземпляров и операторов. Например, класс, структура или интерфейс могут содержать несколько объявлений методов с одним именем с условием, что данные объявления методов отличаются по сигнатуре (§3.6). Обратите внимание, что базовые классы не размещаются в области объявления класса, а также базовые интерфейсы не размещаются в области объявления интерфейса. Таким образом производный класс или интерфейс могут содержать объявление члена с одинаковым именем в качестве унаследованного члена. Такой член скрывает унаследованный член.
* Каждое объявление делегата создает новую область объявления. Имена размещаются в данной области объявления посредством формальных параметров (фиксированных параметров и массивов параметров) и параметров типа.
* Каждое объявление перечисляемого типа создает новую область объявления. Имена размещаются в данной области объявления посредством объявлений членов перечисляемых типов.
* Каждое объявление метода, объявление индексатора, объявление оператора, объявление конструктора экземпляров и анонимная функция создает новую область объявления, которая называется областью объявлений локальных переменных. Имена размещаются в данной области объявления посредством формальных параметров (фиксированных параметров и массивов параметров) и параметров типов. Предполагается, что тело члена функции или анонимной функции (при наличии) вложено в область объявлений локальных переменных. Наличие в области объявления локальных переменных и вложенной области объявления локальных переменных элементов с одинаковым именем является ошибкой. Таким образом, в рамках вложенной области объявления невозможно объявить локальную переменную или константу с одинаковым именем в родительской области объявления. Две области объявления могут содержать элементы с одинаковыми именами, если ни одна из областей не содержит другую.
* Каждый блок или блок switch, а также операторы *for*, *foreach* и *using* создают область объявлений для локальных переменных и локальных констант . Имена размещаются в данной области объявления посредством объявлений локальных переменных и объявлений локальных констант. Обратите внимание, что блоки, используемые в качестве тела члена функции или анонимной функции (или внутри такого тела), вложены в область объявлений локальных переменных, объявленной этими функциями для своих параметров. Следовательно, наличие, к примеру, метода с одинаковыми именами локальной переменной и параметра является ошибкой.
* Каждый блок или блок switch создает отдельную область объявления для меток. Имена размещаются в данной области объявления посредством помеченных операторов, а ссылки на имена осуществляются посредством операторов goto. Область объявления меток блока содержит все вложенные блоки. Таким образом в рамках вложенного блока невозможно объявить метку с одинаковым именем метки в родительском блоке.

Текстовый порядок, в котором имена объявляются, в общем случае не имеет значения. В частности, текстовый порядок не имеет значения для объявления и использования пространств имен, констант, методов, свойств, событий, индексаторов, операторов, конструкторов экземпляров, деструкторов, статических конструкторов и типов. Порядок объявления имеет значение в следующих случаях.

* Порядок объявления объявлений полей и локальных переменных определяет порядок выполнения их инициализаторов (при их наличии).
* Локальные переменные должны быть заданы до их использования (§3.7).
* Порядок объявлений членов перечислений (§14.3) важен, если опущены значения константных выражений.

Область объявления пространства имен является «открыто завершенной», поэтому два объявления пространств имен с одинаковыми полными именами размещаются в одной области объявления. Пример

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

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

два объявления пространств имен вверху размещаются в одной и той же области объявлений, объявляя в этом случае два класса с полными именами Megacorp.Data.Customer и Megacorp.Data.Order. Поскольку два объявления размещаются в одной области объявления, если в каждом объявлении содержится объявление класса с одинаковым именем, происходит ошибка времени компиляции.

Согласно указанному выше, область объявления блока содержит любые вложенные блоки. Поэтому в следующем примере методы F и G приводят к ошибке времени выполнения, так как имя i объявлено во внешнем блоке и не может быть повторно объявлено во внутреннем блоке. Однако методы H и I верны, так как два имени i объявлены в отдельных невложенных блоках.

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();  
 }  
}

## Члены

Пространства имен и типы содержат члены. Члены сущности обычно доступны посредством использования полного имени, начинающегося со ссылки на сущность, затем следует маркер ".", а затем имя члена.

Члены типа либо объявляются в объявлении типа, либо наследуются от базового класса типа. Когда тип наследуется из базового класса, все члены базового класса, исключая конструкторы экземпляров, деструкторы и статические конструкторы, становятся членами производного типа. Объявленная доступность члена базового класса не контролирует наследование члена, наследование распространяется на каждый член, не являющийся конструктором экземпляров, статическим конструктором или деструктором. Однако наследованный член может быть недоступен в производном типе, либо по причине объявленной доступности (§3.5.1), либо по причине того, что он скрыт объявлением в самом типе (§3.7.1.2).

### Члены пространства имен

Пространства имен и типы, имеющие родительское пространство имен, являются членами глобального пространства имен. Это непосредственно соответствует именам, объявленным в глобальной области объявления.

Пространства имен и типы, объявленные в рамках пространства имен, являются членами данного пространства имен. Это непосредственно соответствует именам, объявленным в области объявления пространства имен.

Пространства имен не имеют ограничений. Невозможно объявить частные, защищенные или внутренние пространства имен, пространства имен всегда открыто доступны.

### Члены структуры

Членами структуры являются члены, объявленные в структуре, и члены, унаследованные от непосредственного базового класса структуры System.ValueType и косвенного базового класса object.

Члены простого типа непосредственно соответствуют членам типа структуры с псевдонимами по простому типу.

* Члены sbyte являются членами структуры System.SByte.
* Члены byte являются членами структуры System.Byte.
* Члены short являются членами структуры System.Int16.
* Члены ushort являются членами структуры System.UInt16.
* Члены int являются членами структуры System.Int32.
* Члены uint являются членами структуры System.UInt32.
* Члены long являются членами структуры System.Int64.
* Члены ulong являются членами структуры System.UInt64.
* Члены char являются членами структуры System.Char.
* Члены float являются членами структуры System.Single.
* Члены double являются членами структуры System.Double.
* Члены decimal являются членами структуры System.Decimal.
* Члены bool являются членами структуры System.Boolean.

### Члены перечисления

Членами перечисления являются константы, объявленные в перечислении, и члены, унаследованные от непосредственного базового класса перечисления System.Enum и косвенных базовых классов System.ValueType и object.

### Члены класса

Членами класса являются члены, объявленные в классе, и члены, унаследованные от базового класса (исключая класс object, не имеющего базовый класс). Члены, унаследованные из базового класса, включают константы, поля, методы, свойства, события, индексаторы, операторы и типы базового класса, но не включают конструкторы экземпляров, деструкторы и статические конструкторы базового класса. Члены базового класса наследуются вне зависимости от их доступности.

Объявление класса может содержать объявления констант, полей, методов, свойств, событий, индексаторов, операторов, конструкторов экземпляров, деструкторов, статических конструкторов и типов.

Члены object и string непосредственно соответствуют членам типов класса, псевдонимами которых они являются.

* Члены object являются членами класса System.Object.
* Члены string являются членами класса System.String.

### Члены интерфейса

Членами интерфейса являются члены, объявленные в интерфейсе и во всех базовых интерфейсах данного интерфейса. Члены в классе object, строго говоря, не являются членами какого-либо интерфейса (§13.2). Тем не менее, члены в классе object доступны при поиске членов интерфейса любого типа (§7.4).

### Члены массива

Членами массива являются члены, унаследованные из класса System.Array.

### Члены делегата

Членами делегата являются члены, унаследованные из класса System.Delegate.

## Доступ к члену

Объявления членов позволяют осуществлять контроль доступа членов. Доступность члена устанавливается посредством объявленной доступности (§3.5.1) члена в сочетании с доступностью непосредственно содержащем типе, при его наличии.

Когда доступ к определенному члену разрешен, член является доступным. И наоборот, когда доступ к определенному члену запрещен, член является недоступным. Доступ к члену разрешен, когда текстовое положение, в котором происходит доступ, включено в домен доступности (§3.5.2) члена.

### Объявленная доступность

Возможные варианты объявленной доступности члена представлены ниже.

* Открытая, выбираемая путем включения модификатора public в объявление члена. Интуитивное понимание модификатора public — "доступ не ограничен".
* Защищенная, выбираемая путем включения модификатора protected в объявление члена. Интуитивное понимание модификатора protected — "доступ ограничен классом-контейнером или типами, производными от класса-контейнера".
* Внутренняя, выбираемая путем включения модификатора internal в объявление члена. Интуитивное понимание модификатора internal — "доступ ограничен данной программой".
* Внутренняя защищенная (обозначает защищенную или внутреннюю доступность), выбираемая путем включения модификаторов protected и internal в объявление члена. Интуитивное понимание модификатор protected internal — "доступ ограничен данной программой или типами, производными от класса-контейнера".
* Частная, выбираемая путем включения модификатора private в объявление члена. Интуитивное понимание модификатора private — "доступ ограничен типом-контейнером".

В зависимости от контекста, в котором происходит объявление члена, разрешены только определенные типы объявленной доступности. Кроме того, когда объявление члена не включает модификаторы доступа, контекст, в котором происходит объявление, определяет объявленную доступность по умолчанию.

* Пространства имен неявно имеют объявленную доступность public. Модификаторы доступа запрещены для объявлений пространств имен.
* Типы, объявленные в блоках компиляции, или пространства имен могут иметь объявленную доступность public или internal, а по умолчанию используется объявленная доступность internal.
* Члены класса могут иметь любую из пяти типов объявленной доступности, а по умолчанию используется объявленная доступность private. (Обратите внимание, что тип, объявленный в качестве члена класса, может иметь любую из пяти объявленных доступностей, тогда как тип, объявленный в качестве члена пространства имен, может иметь только объявленную доступность public или internal.)
* Члены структуры могут иметь объявленную доступность public, internal или private, по умолчанию используется объявленная доступность private, так как структуры неявно запечатаны. Члены структуры, размещенные в структуре (т. е. не унаследованные данной структурой), не могут иметь статусы объявленной доступности protected или protected internal. (Обратите внимание, что тип, объявленный в качестве члена структуры, может иметь объявленную доступность public, internal или private, тогда как тип, объявленный в качестве члена пространства имен, может иметь только объявленную доступность public или internal.)
* Члены интерфейса неявно имеют объявленную доступность public. Модификаторы доступа запрещены для объявлений члена интерфейса.
* Члены перечисления неявно имеют объявленную доступность public. Модификаторы доступа запрещены для объявлений члена перечисления.

### Домены доступности

Домен доступности члена состоит из (возможно, не связанных) разделов текста программы, в которых разрешен доступ к члену. С целью определения домена доступности члена он считается верхнего уровня, если он не объявлен в рамках типа, и считается вложенным, если он объявлен в рамках другого типа. Дополнительно определяется текст программы как весь текст программы, содержащийся во всех исходных файлах программы, а текст программы типа определяется как весь текст программы, содержащийся в объявлениях типа данного типа (включая, возможно, типы, вложенные в этот тип).

Домен доступности предопределенного типа (например, object, int или double) неограничен.

Домен доступности непривязанного типа верхнего уровня T (§4.4.3), объявленного в программе P, определяется следующим образом.

* Если для T объявлена доступность public, доменом доступности T является текст программы P и любая программа, ссылающаяся на P.
* Если объявленный уровень доступности T — internal, то домен доступности T совпадает с текстом программы P.

Из этих определений следует, что домен доступности свободного типа верхнего уровня всегда является по крайней мере текстом программы, в которой данный тип объявлен.

Доменом доступности для сформированного типа T<A1, ...,AN> является пересечение домена доступности обобщенного свободного типа T и доменов доступности аргументов типа A1, ...,AN.

Домен доступности вложенного члена M, объявленного в типе T в программе P, определяется следующим образом (учитывая, что M может являться типом).

* Если объявленный уровень доступности M — public, то домен доступности M совпадает с доменом доступности T.
* Если для M объявлена доступность protected internal, пусть D будет объединением текста программы P и текста программы любого типа, произведенного из T, объявленного вне P. Доменом доступности M является пересечение домена доступности T с доменом доступности D.
* Если для M объявлена доступность protected, пусть D будет объединением текста программы T и текста программы любого типа, произведенного из T. Доменом доступности M является пересечение домена доступности T с доменом доступности D.
* Если объявленный уровень доступности M — internal, то домен доступности M представляет собой пересечение домена доступности T с текстом программы P.
* Если объявленный уровень доступности M — private, то домен доступности M совпадает с текстом программы T.

Из этих определений следует, что домен доступности вложенного члена всегда является по крайней мере текстом программы, в которой член объявлен. Дополнительно следует, что домен доступности члена никогда не включает больше объектов, чем домен доступности типа, в котором он объявлен.

Интуитивно это значит, что при доступе к типу или члену M для обеспечения разрешения доступа вычисляются следующие шаги.

* Сначала, если M объявлен в типе (в противоположность блоку компиляции или пространству имен), при недоступности данного типа возникает ошибка времени компиляции.
* Затем, если для M объявлена доступность public, доступ разрешается.
* В противном случае, если M объявлен как protected internal, доступ разрешается в том случае, когда он происходит в рамках программы, в которой объявлен M, или когда он происходит в рамках класса, произведенного из класса, в котором объявлен M, и доступ выполняется посредством производного типа класса (§3.5.3).
* В противном случае, если M объявлен как protected, доступ разрешается в том случае, когда он происходит в рамках класса, в котором объявлен M, или когда он возникает в рамках класса, произведенного из класса, в котором объявлен M, и доступ выполняется посредством производного типа класса (§3.5.3).
* В противном случае, если для M объявлена доступность internal, доступ разрешается при условии, что он осуществляется в рамках программы, в которой объявлен M.
* В противном случае, если для M объявлена доступность private, доступ разрешается при условии, что он осуществляется в типе, в котором объявлен M.
* В противном случае тип или член недоступны, и возникает ошибка времени компиляции.

В этом примере

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;  
 }  
}

классы и члены имеют следующие области доступности.

* Домен доступности A и A.X неограничен.
* Доменом доступности A.Y, B, B.X, B.Y, B.C, B.C.X и B.C.Y является текст содержащей их программы.
* Доменом доступности A.Z является текст программы A.
* Доменом доступности B.Z и B.D является текст программы B, включая текст программ B.C и B.D.
* Доменом доступности B.C.Z является текст программы B.C.
* Доменом доступности B.D.X и B.D.Y является текст программы B, включая текст программы B.C и B.D.
* Доменом доступности B.D.Z является текст программы B.D.

Как видно из примера, область доступности члена никогда не превышает области доступности содержащего типа. Например, несмотря на то что все члены X имеют открытую объявленную доступность, все члены, кроме A.X, имеют домены доступности, ограниченные тип-контейнером.

Согласно описанию в разделе §3.4 все члены базового класса, кроме конструкторов экземпляров, деструкторов и статических конструкторов, наследуются производными типами. Это касается даже закрытых членов базового класса. Однако домен доступности закрытого класса включает только текст программы типа, в котором член объявлен. В этом примере

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  
 }  
}

класс B наследует закрытый член x от класса A. Так как член является закрытым, он доступен только в рамках тела класса A. Поэтому доступ к b.x успешно осуществляется в методе A.F, но завершается ошибкой в методе B.F.

### Защищенный доступ для членов экземпляров

При доступе к члену экземпляра protected вне текста программы, в котором он объявлен, и при доступе к члену экземпляра protected internal вне текста программы, в котором он объявлен, доступ должен осуществляться в объявлении класса, производного от класса, в котором он объявлен. Дополнительно доступ должен осуществляться через экземпляр производного типа класса или типа класса сконструированного из него. Данное ограничение предотвращает доступ одного производного класса к закрытым членам другого производного класса, даже если члены унаследованы от одного базового класса.

Пусть B будет классом, в котором объявлен защищенный член экземпляра M, и пусть D будет производным классом B. В рамках тела класса D доступ к M может принять одну из следующих форм.

* Неполное имя типа или первичное выражение формы M.
* Первичное выражение формы E.M с условием, что тип E является T или производным от T классом, где T является типом класса D или типом класса, сконструированным из D.
* Основное выражение формы base.M.

Дополнительно к данным формам доступа производный класс может осуществлять доступ к конструктору защищенных экземпляров базового класса в инициализаторе конструктора (§10.11.1).

В этом примере

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  
 }  
}

в рамках A можно осуществить доступ к x через экземпляры A и B, так как в любом случае доступ осуществляется через экземпляр A или производный от A класс. Однако в рамках B невозможно осуществить доступ к x через экземпляр A, так как A не произведен из B.

В этом примере

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";  
 }  
}

три присваивания x разрешены, так как они все осуществляются через экземпляры типов класса, сконструированные из универсального типа.

### Ограничения доступности

Некоторым конструкциям языка C# требуется, чтобы тип был хотя бы доступен как член или другой тип. Считается, что доступность типа T не меньше доступности члена или типа M, если домен доступности T является множеством домена доступности M. Другими словами, доступность T не меньше доступности M, если T доступен во всех контекстах, в которых доступен M.

Существуют следующие ограничения доступности.

* Прямой базовый класс типа класса должен быть не менее доступен, чем сам тип класса.
* Явный базовый интерфейс типа интерфейса должен быть не менее доступен, чем сам тип интерфейса.
* Тип возвращаемого значения и типы параметров типа делегата должны быть не менее доступны, чем сам тип делегата.
* Тип константы должен быть не менее доступен, чем сама константа.
* Тип поля должен быть не менее доступен, чем само поле.
* Тип возвращаемого значения и типы параметров метода должны быть не менее доступны, чем сам метод.
* Тип свойства должен быть не менее доступен, чем само свойство.
* Тип события должен быть не менее доступен, чем само событие.
* Тип и типы параметра индексатора должны быть не менее доступны, чем сам индексатор.
* Тип возвращаемого значения и типы параметра оператора должны быть не менее доступны, чем сам оператор.
* Типы параметра конструктора экземпляров должны быть не менее доступны, чем сам конструктор экземпляров.

В этом примере

class A {...}

public class B: A {...}

класс B приводит к ошибке времени компиляции, так как A менее доступен, чем B.

Аналогично, в примере

class A {...}

public class B  
{  
 A F() {...}

internal A G() {...}

public A H() {...}  
}

метод H в B приводит к ошибке времени компиляции, так как тип возвращаемого значения A менее доступен, чем метод.

## Сигнатуры и перегрузка

Методы, конструкторы экземпляров, индексаторы и операторы характеризуются их сигнатурами.

* Сигнатура метода состоит из имени метода, числа параметров типа и типа и вида (значение, ссылка или выход) каждого ее параметра, учтенного в порядке слева направо. Для данных целей каждый параметр типа метода, возникающий в типе формального параметра, идентифицируется не по имени, а по порядковому номеру в списке аргументов типа метода. Сигнатура метода специально не содержит ни тип возвращаемого значения, ни модификатор params, который может быть указан для самого правого параметра, и не содержит дополнительных ограничений параметра типа.
* Сигнатура конструктора экземпляра состоит из типа и вида (значение, ссылка или выход) каждого из формальных параметров, учтенного в порядке слева направо. Сигнатура конструктора экземпляра специально не содержит модификатор params, который может быть указан для самого правого параметра.
* Сигнатура индексатора состоит из типа каждого его формального параметра, учтенного в порядке слева направо. Сигнатура индексатора специально не содержит тип элемента и не содержит модификатор params, который может быть указан для самого правого параметра.
* Сигнатура оператора состоит из имени оператора и типа каждого его формального параметра, учтенного в порядке слева направо. Сигнатура оператора специально не содержит тип результата.

Сигнатуры обеспечивают использование механизма перегрузки членов в классах, структурах и интерфейсах.

* Перегрузка методов позволяет классу, структуре или интерфейсу объявить несколько одноименных методов при условии, что их сигнатуры уникальны в рамках класса, структуры или интерфейса.
* Перегрузка конструкторов экземпляров позволяет классу или структуре объявить несколько конструкторов экземпляров при условии, что их сигнатуры уникальны в рамках класса или структуры.
* Перегрузка индексаторов позволяет классу, структуре или интерфейсу объявить несколько одноименных индексаторов при условии, что их сигнатуры уникальны в рамках класса, структуры или интерфейса.
* Перегрузка операторов позволяет классу или структуре объявить несколько одноименных операторов при условии, что их сигнатуры уникальны в рамках класса или структуры.

Несмотря на то что модификаторы параметров out и ref считаются частью сигнатуры, члены, объявленные в одном типе, не могут отличаться по сигнатуре только параметрами ref и out. Если два члена, объявленные в одном типе, будут иметь одинаковые сигнатуры, в которых все параметры в обоих методах с модификаторами out изменены на модификаторы ref, возникает ошибка времени компиляции. Для других целей сопоставления сигнатур (например, для скрытия или переопределения) ref и out учитываются в качестве части сигнатуры и не соответствуют друг другу. (Данное ограничение позволяет простым способом транслировать программы на C# для выполнения в инфраструктуре Common Language Infrastructure (CLI), не обеспечивающей способ определения методов, отличающихся исключительно по ref и out.)

Для сигнатур типы object и dynamic являются идентичными. Члены, объявленные в одном типе, могут не отличаться в сигнатурах одними лишь object и dynamic.

В следующем примере представлен набор объявлений перегруженных методов с их сигнатурами.

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  
}

Обратите внимание, что модификаторы параметров ref и out (§10.6.1) являются частью сигнатуры. Поэтому F(int) и F(ref int) являются уникальными сигнатурами. Однако F(ref int) и F(out int) не могут быть объявлены в одном интерфейсе, так как их сигнатуры отличаются только модификаторами ref и out. Также обратите внимание, что тип возвращаемого значения и модификатор params не являются частью сигнатуры, поэтому невозможно выполнить перегрузку только на основе типа возвращаемого значения или на основе включения или исключения модификатора params. По этим причинам представленные выше объявления методов F(int) и F(params string[]) приводят к ошибке времени компиляции.

## Области видимости

Областью видимости имени является часть текста программы, в рамках которой можно ссылаться на сущность, объявленную именем без квалификации имени. Области могут быть вложенными, а внутренняя область может повторно объявлять значение имени из внешней области (однако это не снимает ограничение, налагаемое разделом §3.3, что в рамках вложенного блока невозможно объявить локальную переменную с именем, совпадающим с именем локальной переменной во внешнем блоке). Имя из внешней области при этом считается скрытым в регионе текста программы, покрываемом внутренней областью, а доступ к внешнему имени возможен только по квалифицирующему имени.

* Областью члена пространства имен, объявленного объявлением члена пространства имен (§9.5), без вмещающего объявления пространства имен является полный текст программы.
* Областью члена пространства имен, объявленного объявлением члена пространства имен в рамках объявления пространства имен с полным именем N, является тело пространства имен каждого объявления пространства имен с полным именем N, или начинающимся с N и последующей точкой.
* Область имени, определенного директивой extern alias, распространяется на директивы using, глобальные атрибуты и объявления членов пространства имен незамедлительно содержащегося блока компиляции или тела пространства имен. Директива extern alias не вносит какие-либо новые члены в нижестоящую область объявления. Иначе говоря, директива extern alias не является транзитивной, и влияет только на единицу компиляции или тело пространства имен, в котором находится.
* Область имени, заданная или импортированная с помощью директивы using (§9.4), распространяется на объявления членов пространства имен блока компиляции или тела пространства имен, в котором содержится директива using. Директива using может привести к образованию нуля или более пространств имен или типов имен, доступных в рамках определенного блока компиляции или тела пространства имен, но не может привести к созданию новых членов в нижестоящей области объявления. Другими словами, директива using не является транзитивной, а скорее влияет только на блок компиляции или тело пространства имен, в котором находится.
* Областью параметра типа, объявленного списком параметров типа в объявлении класса (§10.1), является база класса, предложения ограничений параметров типа и тело класса данного объявления класса.
* Областью параметра типа, объявленного списком параметров типа в объявлении структуры (§11.1), является интерфейс структуры, предложения ограничений параметров типа и тело структуры данного объявления структуры.
* Областью параметра типа, объявленного списком параметров типа в объявлении интерфейса (§13.1), является база интерфейса, предложения ограничений параметров типа и тело интерфейса данного объявления интерфейса.
* Областью параметра типа, объявленного списком параметров типа в объявлении делегата (§15.1), является тип возвращаемого значения, список формальных параметров и предложения ограничений параметров типа данного объявления делегата.
* Областью члена, объявленного объявлением члена класса (§10.1.6), является тело класса, в котором сделано объявление. Кроме того, область члена класса распространяется на тела класса производных классов, включенных в домен доступности (§3.5.2) члена.
* Областью члена, объявленного объявлением члена структуры (§11.2), является тело структуры, в котором сделано объявление.
* Областью члена, объявленного объявлением члена перечисляемого типа (§14.3), является тело перечисляемого типа, в котором сделано объявление.
* Областью параметра, объявленного в объявлении метода (§10.6), является тело метода данного объявления метода.
* Областью параметра, объявленного в объявлении индексатора (§10.9), являются объявления метода доступа данного объявления индексатора.
* Областью параметра, объявленного в объявлении оператора (§10.10), является блок данного объявления оператора.
* Областью параметра, объявленного в объявлении конструктора (§10.11), является инициализатор конструктора и блок данного объявления конструктора.
* Областью параметра, объявленного в лямбда-выражении (§7.15), является тело лямбда-выражения данного лямбда-выражения.
* Областью параметра, объявленного в выражении анонимного метода (§7.15), является блок данного выражения анонимного метода.
* Областью метки, объявленной в помеченном операторе (§8.4), является блок, в котором содержится объявление.
* Областью видимости локальной переменной, описанной в объявлении локальной переменной (§8.5.1), является блок, в котором встречается объявление.
* Областью локальной переменной, объявленной в блоке switch оператора switch (§8.7.2), является блок switch.
* Областью локальной переменной, объявленной в инициализаторе for оператора for (§8.8.3), является инициализатор for, условие for, итератор for и содержащий оператор оператора for.
* Областью локальной константы, объявленной в объявлении локальной константы (§8.5.2), является блок, в котором содержится объявление. Ссылка на локальную константу в текстовой позиции, предшествующей декларатору константы, является ошибкой времени компилирования.
* Область переменной, объявленной в качестве части оператора foreach, оператора using, оператора lock или выражения запроса, определяется путем расширения заданной конструкции.

В рамках области члена пространства имен, класса, структуры или перечисляемого типа можно ссылаться на членов в текстовой позиции, предшествующей объявлению члена. Пример

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

int i = 0;  
}

В данном примере ссылка F на i до объявления является верной.

В рамках области локальной переменной ссылка на локальную переменную в текстовой позиции, предшествующей декларатору локальной переменной локальной переменной, будет являться ошибкой времени компилирования. Пример

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  
 }  
}

В вышеуказанном методе F первое присваивание i явно не ссылается на поле, объявленное во внешней области. Однако существует ссылка на локальную переменную, что приводит к ошибке времени компилирования, так как текстуальность предшествует объявлению переменной. В методе G использование j в инициализаторе для объявления j является верным, так как использование не предшествует декларатору локальной переменной. В методе H последующий декларатор локальной переменной верно ссылается на локальную переменную, объявленную в более раннем деклараторе локальной переменной в рамках того же декларатора локальной переменной.

Правила назначения областей для локальных переменных разработаны для того, чтобы значение имени в контексте выражения использовалось только в рамках блока. Если область локальной переменной расширена только из объявления до конца блока, в вышеуказанном примере первое присваивание будет присвоено переменной экземпляра, второе присваивание будет присвоено локальной переменной, что, возможно, приведет к ошибкам времени компилирования, если операторы блока будут затем переупорядочены.

Значение имени в рамках блока может отличаться в зависимости от контекста, в котором используется имя. В этом примере

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"  
 }  
}

имя A используется в контексте выражения для ссылки на локальную переменную A и в контексте типа для ссылки на класс A.

### Скрытие имени.

Область сущности обычно заключает больше текста программы, чем область объявления сущности. В частности, область сущности может включать объявления, представляющие новые области объявлений, содержащие сущности с одинаковыми именами. Такие объявления приводят к тому, что исходная сущность становится скрытой. И наоборот, сущность считается видимой, если она не скрыта.

Скрытие имени происходит, когда области перекрываются через вложение и когда области перекрываются через наследование. Характеристики двух типов скрытия описываются в следующих разделах.

#### Скрытие через вложение.

Скрытие имени через вложение возникает в результате вложения пространств имен или типов в рамках пространств имен, в результате вложения типов в рамках классов или структур и в результате объявлений параметров и локальных переменных.

В этом примере

class A  
{  
 int i = 0;

void F() {  
 int i = 1;  
 }

void G() {  
 i = 1;  
 }  
}

в рамках метода F переменный экземпляр i скрыт локальной переменной i, но в рамках метода G экземпляр i все еще ссылается на переменный экземпляр.

Когда имя во внутренней области скрывает имя внешней области, скрываются все перегруженные вхождения данного имени. В этом примере

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) {}  
 }  
}

вызов F(1) приводит к вызову F, объявленному в Inner, так как все внешние вхождения F скрыты внутренним объявлением. По той же причине вызов F("Hello") приводит к ошибке времени компиляции.

#### Скрытие через наследование

Скрытие имени через наследование происходит, когда классы или структуры повторно объявляют имена, унаследованные из базовых классов. Этот тип скрытия имени принимает одну из следующих форм.

* Константа, поле, свойство, событие или тип, представленные в классе или структуре, скрывают все члены базового класса с таким же именем.
* Метод, представленный в классе или структуре, скрывает все члены базового класса, не являющиеся методами, и все методы базового класса с такой же сигнатурой (имя метода и попадания, модификаторы и типы параметра).
* Индексатор, представленный в классе или структуре, скрывает все индексаторы базового класса с такой же сигнатурой (попадания и типы параметра).

Правила, управляющие объявлениями операторов (§10.10), делают невозможным для производного класса объявление оператора с сигнатурой, аналогичной сигнатуре оператора базового класса. Поэтому операторы никогда не скрывают друг друга.

В противоположность скрытию имени из внешней области, скрытие доступного имени из унаследованной области приводит к предупреждению. В этом примере

class Base  
{  
 public void F() {}  
}

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

объявление F в классе Derived приводит к выводу предупреждения. Скрытие наследуемого имени не является ошибкой, поскольку это препятствует отдельному развитию базовых классов. Например, вышеуказанная ситуация могла наступить, так как более поздняя версия Base представила метод F, который не был представлен в более ранней версии класса. Если бы вышеупомянутая ситуация была ошибочной, любое изменение, сделанное для базового класса в библиотеке класса отдельной версии, могло потенциально привести к недопустимости производных классов.

Предупреждение, вызванное скрытием унаследованного имени, можно устранить с помощью модификатора new:

class Base  
{  
 public void F() {}  
}

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

Модификатор new указывает, что метод F в классе Derived является "новым" и скрытие унаследованного члена выполняется намеренно.

Объявление нового члена приводит к скрытию унаследованного члена только в рамках области нового члена.

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  
}

В вышеуказанном примере объявление F в классе Derived приводит к скрытию метода F, унаследованного от Base, но, так как для нового метода F в классе Derived объявлен закрытый доступ, его область не распространяется на класс MoreDerived. Поэтому вызов F() в классе MoreDerived.G является допустимым и приводит к вызову Base.F.

## Имена пространств имен и типов

Некоторые контексты в программе на C# требуют указания имени пространства имен или имени типа.

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-listoptqualified-alias-member

Имя пространства имен — это имя пространства имен или типа, ссылающееся на пространства имен. Согласно описанию ниже следующее разрешение имени пространства имен или типа от имени пространства имен должно ссылаться на пространство имен, в противном случае возникает ошибка времени компилирования. Аргументы типов (§4.4.1) не могут быть представлены в имени пространства имен (только типы с аргументами типа).

Имя типа — это имя пространства имен или типа, ссылающееся на тип. Согласно описанию ниже следующее разрешение имени пространства имен или типа от имени типа должно ссылаться на тип, в противном случае возникает ошибка времени компилирования.

Если имя пространства имен или типа является членом квалифицированного псевдонима, его значение соответствует описанию в разделе §9.7. В противном случае имя пространства имен или типа имеет одну из следующих четырех форм.

* I
* I<A1, ..., AK>
* N.I
* N.I<A1, ..., AK>

где I — отдельный идентификатор, N — имя пространства имени или типа и <A1, ..., AK> — дополнительный список аргументов типа. Если список аргументов типа не указан, K считается равным нулю.

Значение имени пространства имен или типа определяется следующим образом.

* Имя пространства имен или типа имеет одну из форм I или форму I<A1, ..., AK>:
* Если K имеет нулевое значение, имя пространства имен или типа появляется в универсальном методе объявления (§10.6), а также если это объявление включает параметр типа (§10.1.3) с именем I, имя пространства имен или типа ссылается на данный параметр типа.
* В противном случае, если имя пространства имен или типа появляется в объявлении типа, то для каждого типа экземпляра T (§10.3.1), начиная с типа экземпляра данного объявления типа и далее для типов экземпляра каждого включающего класса или объявления структуры (при наличии):
* Если K имеет нулевое значение, а объявление T включает параметр типа с именем I, то имя пространства имен или типа ссылается на данный параметр типа.
* В противном случае, если имя пространства имен или типа появляется в теле объявления типа, а T или любой его базовый тип содержит вложенный доступный тип с именем I и параметрами типа K, то имя пространства имен или типа ссылается на этот тип, сформированный с данными аргументами типа. При наличии более одного типа структуры выбирается тип, объявленный в рамках большего производного типа. Обратите внимание, что члены, не являющиеся типами (константы, поля, методы, свойства, индексаторы, операторы, экземпляры, конструкторы, деструкторы и статические конструкторы), и члены типа с различным числом параметров типа игнорируются при определении значения имени пространства имен или типа.
* Если предыдущие шаги не были успешно выполнены, для каждого пространства имен N, начиная с пространства имен, содержащего имя пространства имен или типа, далее для каждого включающего пространства имен (при наличии), и заканчивая глобальным пространством имен, вычисляются следующие шаги до нахождения сущности.
* Если K равно нулю и I является именем пространства имен в N, то:
* Если местоположение, содержащее имя пространства имен или типа, заключено объявлением пространства имен для N, а объявление пространства имен содержит директиву extern alias или директиву using-alias, связывающую имя I с пространством имен или типом, то имя пространства имен или типа неоднозначно, поэтому возникает ошибка времени компилирования.
* В противном случае имя пространства имен или типа ссылается на пространство имен с именем I в N.
* Иначе, если N содержит доступный тип с именем I и параметрами типа K, то:
* Если K имеет нулевое значение и местоположение, содержащее имя пространства имен или типа, заключено объявлением пространства имен для N, а объявление пространства имен содержит директиву extern alias или директиву using alias, связывающие имя I с пространством имен или типом, то имя пространства имен или типа неоднозначно, поэтому возникает ошибка времени компилирования.
* Иначе имя пространства имен или типа относится к типу, сформированному с данными аргументами типа.
* Наоборот, местоположение, содержащее имя пространства имен или типа, заключено объявлением пространства имен для N.
* Если K имеет нулевое значение и объявление пространства имен содержит директиву extern alias или директиву using alias, связывающих имя I с импортированным пространством имен или типом, то имя пространства имен или типа ссылается на данное пространство имен или тип.
* В противном случае, если пространства имен, импортированные с помощью директивы using namespace, объявления пространства имен содержат точно один тип с именем I и параметрами типа K, то имя пространства имен или типа ссылается на данный тип, сконструированный с заданными аргументами типа.
* В противном случае, если пространства имен, импортированные с помощью директивы using namespace, объявления пространства имен содержат точно один тип с именем I и параметрами типа K, то имя пространства имен или типа является неоднозначным и возникает ошибка.
* В противном случае имя пространства имен или типа не определено, поэтому возникает ошибка времени компилирования.
* В противном случае имя пространства имен или типа имеет одну из форм N.I или форму N.I<A1, ..., AK>. N впервые разрешается в качестве имени пространства имен или типа. Если разрешение N неуспешно, возникает ошибка времени компилирования. В противном случае N.I или N.I<A1, ..., AK> разрешается следующим образом.
* Если K имеет нулевое значение и N ссылается на пространство имен и N содержит вложенное пространство имен с именем I, то имя пространства имен или типа ссылается на данное вложенное пространство имен.
* В противном случае, если N ссылается на пространство имен и N содержит доступный тип с именем I и параметрами типа K, то имя пространства имен или типа ссылается на данный тип, сконструированный с заданными аргументами типа.
* В противном случае, если N ссылается на тип класса (возможно, сформированного) или тип структуры и если N или любой его базовый класс содержит вложенный доступный тип с именем I и параметрами типа K, то имя пространства имен или типа ссылается на данный тип, сформированный с заданными аргументами типа. При наличии более одного типа структуры выбирается тип, объявленный в рамках большего производного типа. Следует отметить, что если значение N.I определяется как часть разрешения спецификации базового класса N, прямой базовый класс N считается объектом (§10.1.4.1).
* В противном случае N.I является недействительным именем пространства имен или типа, поэтому возникает ошибка времени компилирования.

Ссылка имени пространства имен или типа на статический класс (§10.1.1.3) допускается только в следующих случаях.

* Имя пространства имен или типа T в имени пространства имен или типа формы T.I или
* Имя пространства имен или типа равно T в выражении typeof (§7.5.11) в виде typeof(T).

### Полные имена

Каждое пространство имен и тип имеют полные имена, уникально идентифицирующие пространство имен или тип среди других пространств имен и типов. Значение полного имени пространства имен или типа N определяется следующим образом.

* Если N является членом глобального пространства имен, его полное имя N.
* В противном случае его полное имя S.N, где S — полное имя пространства имен или типа, в котором N объявлен.

Другими словами, полное имя N представляет полный иерархический путь идентификаторов, ведущих к N, начиная с глобального пространства имен. Так как каждый член пространства имен или типа должен иметь уникальное имя, полное имя пространства имен или типа всегда уникально.

В следующем примере представлено несколько объявлений пространств имен и типов вместе с соответствующими полными именами.

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  
}

## Автоматическое управление памятью

C# развертывает автоматическое управление памятью, что освобождает разработчиков от ручного выделения и освобождения памяти, занятой объектами. Политики автоматического управления памятью реализованы в сборщике мусора. Далее представлен жизненный цикл объекта управления памятью.

1. Когда объект создается, для него выделяется память, выполняется конструктор и объект считается существующим.
2. Если объект или его любая часть не могут быть вызваны посредством любой возможной продолжительности выполнения, отличной от выполнения деструкторов, объект считается неиспользуемым и становится доступным для деструкции. Компилятор C# и сборщик мусора могут анализировать код с целью определения того, какие ссылки на объект могут быть использованы в будущем. Например, если локальная переменная в области является единственной существующей ссылкой на объект, но на нее никогда не ссылались в любом возможном предложении выполнения, начиная с текущего момента выполнения в процедуре, сборщик мусора может (но не обязательно) посчитать данный объект неиспользуемым.
3. После того, как объект становится доступным для деструкции, через некоторое указанное время выполняется деструктор (§10.13) (при наличии) для объекта. В обычных ситуациях деструктор выполняется для объекта только один раз, хотя связанные с реализацией API могут разрешать переопределить это поведение.
4. Если после выполнения деструктора для объекта данный объект или любая его часть не доступны для любого продолжения выполнения, включая выполнение деструкторов, объект считается недоступным и становится доступным для коллекции.
5. В итоге, через некоторое время после того, как объект стал доступным для коллекции, сборщик мусора очищает память, связанную с объектом.

Сборщик мусора хранит информацию об использовании объекта и использует ее для принятия решений управления памятью, например, где расположить в памяти новый созданный объект, когда следует переместить объект и когда объект больше не используется или недоступен.

Подробно другим языкам, допускающим наличие сборщика мусора, C# разработан таким образом, чтобы сборщик мусора мог реализовывать широкий спектр политик управления памятью. Например, C# не требует, чтобы осуществлялись выполнение деструкторов или подборка объектов сразу после того, как они становятся доступными, или выполнение деструкторов в любом определенном порядке или в любом определенном потоке.

Поведение сборщика мусора можно в некоторой степени контролировать посредством статических методов класса System.GC. Этот класс может использоваться для запроса выполнения сборки, выполнения деструкторов (или отмены выполнения) и т. п.

Так как сборщику мусора предоставляются широкие возможности в решении момента сборки объектов и выполнения деструкторов, соответствующая реализация может привести к результату, отличающемуся от представленного кода. Программа

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();  
 }  
}

приводит к созданию экземпляра класса A и экземпляра класса B. Данные объекты становятся доступны сборщику мусора, когда переменной b присваивается значение null, так как после этого доступ к ним невозможен посредством любого пользовательского кода. Результат может быть одним из следующих:

Destruct instance of A  
Destruct instance of B

или

Destruct instance of B  
Destruct instance of A

так как язык не налагает ограничений на порядок, в котором объекты обрабатываются сборщиком мусора.

В особых случаях отличие между «доступен для деструкции» и «доступен для сборки» может быть очень важным. Например,

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");  
 }  
}

В вышеуказанной программе, если сборщик мусора выбирает выполнение деструктора A до деструктора B, то результат программы может быть следующим:

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

Обратите внимание, что, несмотря на то, что экземпляр A не был использован, а деструктор A был выполнен, методы A (в данном случае F) могут быть все равно вызваны из другого деструктора. Также обратите внимание, что выполнение деструктора может привести к тому, что объект станет снова доступен для использования из основной программы. В этом случае выполнение деструктора B привело к тому, что экземпляр A, который ранее не был использован, стал доступен из существующей ссылки Test.RefA. После вызова WaitForPendingFinalizers экземпляр B доступен для сборки мусора, а экземпляр A недоступен из-за ссылки Test.RefA.

Чтобы избежать несогласованности и неожиданного поведения, для деструкторов рекомендуется выполнять только очистку данных, сохраненных в их собственных полях объектов, и не выполнять действий со ссылочными объектами или статическими полями.

Альтернативой использованию деструкторов может служить реализация классом интерфейса System.IDisposable. Это позволит клиенту объекта определить момент освобождения ресурсов объекта, что обычно происходит путем вызова объекта в качестве ресурса с помощью оператора using (§8.13).

## Порядок выполнения

Выполнение программы на языке C# выполняется таким образом, что побочные эффекты каждого выполняемого потока сохраняются в критических точках выполнения. Побочный эффект — это чтение или запись поля с модификатором volatile, запись в переменную без модификатора volatile, запись во внешний ресурс и передача исключения. Критические точки выполнения, в которых сохраняется порядок данных побочных эффектов, ссылаются на поля с модификаторами volatile (§10.5.3), операторы lock (§8.12) и создание и завершение потока. Среда выполнения может изменить порядок выполнения программы на C#, учитывая следующие ограничения.

* Зависимости данных сохраняются в рамках потока выполнения. То есть значение каждой переменной рассчитывается таким образом, как было бы рассчитано, если бы все операторы в потоке выполнялись в исходном программном порядке.
* Правила упорядочивания инициализации сохраняются (§10.5.4 и §10.5.5).
* Упорядочивание побочных эффектов сохраняется с учетом чтения и записи модификатора volatile (§10.5.3). Кроме того, среда выполнения не должна вычислять часть выражения, если невозможно определить, что значение данного выражения не используется, и что необходимые побочные эффекты не имели места (включая вызванное при вызове метода или доступе к полю с модификатором volatile). При прерывании выполнения программы асинхронным событием (например, при передаче исключения другим потоком) оригинальный программный порядок отображения наблюдаемых побочных эффектов не гарантируется.

# Типы

В языке C# представлены две основные категории типов: типы значений и ссылочные типы. Оба типа значения и ссылочные типы могут использоваться как универсальные типы, принимающие один или несколько параметров типа. Параметры типа могут обозначать как типы значений, так и ссылочные типы.

type:  
value-type  
reference-type   
type-parameter

Типы третьей категории — указатели — доступны только в небезопасном коде. Дополнительные сведения см. далее в разделе §18.2.

Различие между типами значений и ссылочными типами заключается в том, что переменные первого типа непосредственно содержат данные, а переменные второго типа хранят ссылки на соответствующие данные (объекты). Две переменные ссылочного типа могут ссылаться на один объект. Это позволяет изменять объект, на который ссылается одна переменная, выполняя соответствующие операции с другой. Каждая переменная типа значений содержит собственную копию данных. В связи с этим операции с одной переменной не влияют на другую.

В C# применяется унифицированная система типов, в которой значение любого типа может обрабатываться как объект. В C# каждый тип прямо или косвенно наследует от типа класса object, и класс object является первичным базовым классом для всех типов. Обработка значений ссылочного типа как объектов выполняется посредством рассмотрения их как значений типа object. Значения типа значений обрабатываются как объекты с использованием операций упаковки и распаковки (§4.3).

## Типы значений

Существуют следующие типы значений: тип структуры и перечисляемый тип. В C# представлен набор предопределенных типов структуры, называемых простыми типами. Простые типы обозначаются зарезервированными словами.

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

В отличие от переменных ссылочного типа, переменные типа значений могут содержать null только в том случае, если значение имеет обнуляемый тип. Для каждого необнуляемого типа значений существует соответствующий обнуляемый тип, включающий те же значения и значение null.

Когда переменной назначается значение типа значения, создается копия присвоенного значения. В этом заключается отличие от переменных ссылочного типа, при присвоении которых копируются ссылки, а не определяемые ими объекты.

### Тип System.ValueType

Все типы значений неявно наследуются от класса System.ValueType, который, в свою очередь, наследуется от класса object. Получение типов от типов значений не поддерживается. Таким образом, типы значений являются неявно запечатанными (§10.1.1.2).

Обратите внимание, что тип System.ValueType не является типом значения. Он является типом класса, от которого автоматически получаются все типы значений.

### Конструкторы по умолчанию

Во всех типах значений неявно объявляется не содержащий параметров открытый конструктор экземпляров, называемый конструктором по умолчанию. Конструктор по умолчанию возвращает заполненный нулями экземпляр типа значений, называемый значением по умолчанию.

* Для всех простых типов значение по умолчанию представляет собой битовый шаблон, заполненный нулями:
* Для типов sbyte, byte, short, ushort, int, uint, long и ulong значением по умолчанию является 0.
* Для char значение по умолчанию — '\x0000'.
* Для float значение по умолчанию — 0.0f.
* Для double значение по умолчанию — 0.0d.
* Для decimal значение по умолчанию — 0.0m.
* Для bool значение по умолчанию — false.
* Для перечисляемого типа E значение по умолчанию составляет 0 и преобразуется к типу E.
* Для типа структуры значение по умолчанию формируется путем присвоения значений по умолчанию всем полям, имеющим тип значений, а всем полям ссылочного типа — значения null.
* Для обнуляемого типа значение по умолчанию представляет собой экземпляр, свойство HasValue которого имеет значение false, а свойство Value не определено. Для обнуляемых типов значение по умолчанию также называется пустым значением.

Как и любой другой конструктор экземпляров, конструктор по умолчанию для типа значений вызывается с помощью оператора new. В целях повышения эффективности в реализации не обязательно создается вызов конструктора. В приведенном ниже примере выполняется инициализация переменных i и j нулевыми значениями.

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

Поскольку для каждого типа значений неявно определяется не содержащий параметров открытый конструктор экземпляров, в типе структуры не допускается явное объявление не содержащего параметров конструктора. Тем не менее, в типе структуры допускается объявление параметризованных конструкторов экземпляров (§11.3.8).

### Типы структуры

Тип структуры представляет собой тип значений, в котором допускается объявление констант, полей, методов, свойств, индексаторов, операторов, конструкторов экземпляров, статических конструкторов и вложенных типов. Дополнительные сведения об объявлении типов структуры см. в разделе §11.1.

### Простые типы

В C# представлен набор предопределенных типов структуры, называемых простыми типами. Простые типы обозначаются зарезервированными словами. Эти слова представляют собой псевдонимы предопределенных типов структуры в пространстве имен System (см. таблицу ниже).

|  |  |
| --- | --- |
| **Зарезервированное слово** | **Тип с псевдонимом** |
| 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 |

Поскольку простой тип представляет собой псевдоним типа структуры, каждый простой тип содержит члены. Например, int содержит члены, объявленные в System.Int32, а также члены, унаследованные от System.Object. При этом допускаются следующие операторы:

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

Различие между простыми типами и типами структуры заключается в том, что в простых типах допускаются некоторые дополнительные операции:

* Для большинства простых типов допускается создание значений путем записи литералов (§2.4.4). Например, 123 представляет собой литерал типа int, тогда как 'a' — литерал типа char. В C# не предусматривается использование литералов для типов структуры. Создание значений не по умолчанию для других типов структуры практически во всех случаях осуществляется с использованием конструкторов экземпляров соответствующих типов.
* Значение выражения, все операнды которого являются константами простого типа, может быть вычислено на этапе компиляции. Такие выражения называются константными выражениями (§7.19). Выражения, которые содержат операторы, определяемые другими типами структуры, не являются константными.
* Для объявления констант простых типов используется объявление const (§10.4). Константы других типов структуры не поддерживаются, однако аналогичное действие имеют поля static readonly.
* При вычислении операторов преобразования, определенных другими типами структуры, могут использоваться преобразования простых типов. Однако пользовательские операторы преобразования никогда не используются при вычислении другого пользовательского оператора (§6.4.3).

### Целочисленные типы

В C# поддерживается девять целых типов: sbyte, byte, short, ushort, int, uint, long, ulong и char. Целые типы имеют следующие размеры и диапазоны значений:

* Тип sbyte представляет 8-разрядные целые числа со знаком в диапазоне от –128 до 127.
* Тип byte представляет 8-разрядные целые числа без знака в диапазоне от 0 до 255.
* Тип short представляет 16-разрядные целые числа со знаком в диапазоне от –32768 до 32767.
* Тип ushort представляет 16-разрядные целые числа без знака в диапазоне от 0 до 65535.
* Тип int представляет 32-разрядные целые числа со знаком в диапазоне от –2147483648 до 2147483647.
* Тип uint представляет 32-разрядные целые числа без знака в диапазоне от 0 до 4294967295.
* Тип long представляет 64-разрядные целые числа со знаком в диапазоне от –9223372036854775808 до 9223372036854775807.
* Тип ulong представляет 64-разрядные целые числа без знака в диапазоне от 0 до 18446744073709551615.
* Тип char представляет 16-разрядные целые числа без знака в диапазоне от 0 до 65535. Набор возможных значений для типа char соответствует набору символов Юникода. Несмотря на то что представления типов char и ushort совпадают, наборы допустимых операций для каждого типа различаются.

Унарные и бинарные операторы целого типа всегда работают с точностью, соответствующей 32- или 64-разрядным числам со знаком или без знака:

* Для унарных операторов + и ~ операнд преобразуется к типу T, где T — первый тип из набора int, uint, long и ulong, с помощью которого могут быть полностью представлены все возможные значения операнда. Операция выполняется с использованием точности, соответствующей типу T. Результат операции имеет тип T.
* Для унарного оператора – операнд преобразуется к типу T, где T — первый тип из набора int и long, с помощью которого могут быть полностью представлены все возможные значения операнда. Операция выполняется с использованием точности, соответствующей типу T. Результат операции имеет тип T. Не допускается применение унарного оператора – к операндам типа ulong.
* Для бинарных операторов +, –, \*, /, %, &, ^, |, ==, !=, >, <, >= и <= операнды преобразуются к типу T, где T — первый тип из набора int, uint, long и ulong, с помощью которого могут быть полностью представлены все возможные значения обоих операндов. Операция выполняется с использованием точности, соответствующей типу T. Результат операции имеет тип T (или bool для операторов отношения). Для бинарных операторов не допускается использование двух переменных различных типов (например long и ulong).
* Для бинарных операторов << и >> левый операнд преобразуется к типу T, где T — первый тип из набора int, uint, long и ulong, с помощью которого могут быть полностью представлены все возможные значения операнда. Операция выполняется с использованием точности, соответствующей типу T. Результат операции имеет тип T.

Тип char классифицируется как целый тип, однако имеет два отличия от других целых типов:

* Не поддерживается неявное преобразование из других типов к типу char. В частности, несмотря на то что диапазоны значений для типов sbyte, byte и ushort могут быть полностью представлены с помощью типа char, неявное преобразование из типа sbyte, byte или ushort к типу char не существует.
* Константы типа char должны записываться в виде символьных или целочисленных литералов в сочетании с приведением к типу char. Например, (char)10 совпадает с '\x000A'.

Для управления проверкой переполнения при выполнении целочисленных арифметических операций и преобразований используются операторы checked и unchecked (§7.6.12). В контексте checked при переполнении возникает ошибка времени компиляции или порождается исключение System.OverflowException. В контексте unchecked переполнение игнорируется, а все старшие биты, не соответствующие размеру конечного типа, удаляются.

### Типы с плавающей запятой

В C# поддерживается два типа с плавающей запятой: float и double. Типы float и double представляются в 32-разрядном (одинарная точность) или 64-разрядном (двойная точность) формате IEEE 754 и поддерживают следующие наборы значений:

* Положительный и отрицательный нуль. В большинстве случаев поведение значений положительного и отрицательного нуля совпадает. Различие между ними используется лишь в некоторых операторах (§7.8.2).
* Положительная и отрицательная бесконечность. Бесконечность может получиться, например, в результате деления ненулевого значения на нуль. К примеру, в результате операции 1.0 / 0.0 получается положительная бесконечность, а в результате операции –1.0 / 0.0 — отрицательная.
* Нечисловые значения зачастую сокращаются как NaN. Значения NaN получаются в результате выполнения недопустимых операций с плавающей запятой, например при делении нуля на нуль.
* Конечный набор ненулевых значений вида s × m × 2e, где s равно 1 или −1, а значения m и e зависят от конкретного типа с плавающей запятой, выглядит для типа float как 0 < m < 224 и −149 ≤ e ≤ 104, для типа double как 0 < m < 253 и −1075 ≤ e ≤ 970. Допустимыми ненулевыми значениями считаются денормализованные числа с плавающей запятой.

Тип float представляет значения в диапазоне от 1,5 x 10−45 до 3,4 x 1038 (приблизительно) с точностью до 7 знаков.

Тип double представляет значения в диапазоне от 5,0 x 10−324 до 1,7 x 10308 (приблизительно) с точностью до 15–16 знаков.

Если один из операндов бинарного оператора имеет тип с плавающей запятой, второй операнд должен иметь целый тип или тип с плавающей запятой. В этом случае вычисление операции выполняется следующим образом:

* Если один из операторов имеет целый тип, он преобразуется к типу с плавающей запятой, соответствующему типу второго операнда.
* В этом случае, если один из операндов имеет тип double, второй операнд преобразуется к типу double. Операция выполняется с применением точности и диапазона, соответствующих типу double. Результат операции имеет тип double (или bool для операторов отношения).
* В противном случае операция выполняется с применением точности и диапазона, соответствующих типу float. Результат операции имеет тип float (или bool для операторов отношения).

При использовании операторов с плавающей запятой, включая операторы присваивания, никогда не порождаются исключения. Вместо этого в исключительных ситуациях в результате выполнения операции с плавающей запятой получается нуль, бесконечность или нечисловое значение (NaN), как описано ниже:

* Если результат выполнения операции с плавающей запятой слишком мал для конечного формата, результату присваивается значение положительного или отрицательного нуля.
* Если результат выполнения операции с плавающей запятой слишком велик для конечного формата, результату присваивается значение положительной или отрицательной бесконечности.
* Если при выполнении операции с плавающей запятой получается недопустимый результат, операция возвращает результат NaN.
* Если один или оба операнда операции с плавающей запятой имеют значение NaN, операция возвращает результат NaN.

Операции с плавающей запятой могут выполняться с точностью, превышающей точность типа результата операции. Например, в некоторых аппаратных архитектурах поддерживаются типы с плавающей запятой «extended» или «long double», обладающие расширенными диапазоном и точностью по сравнению с типом double. В таких случаях все операции с плавающей запятой неявно выполняются с использованием типа с более высокой точностью. В таких аппаратных архитектурах операции с плавающей запятой выполняются с меньшей точностью только в целях повышения производительности (в случае ее значительного снижения). Чтобы не допустить одновременного снижения производительности и точности при реализации, в C# возможно применение типа с более высокой точностью для выполнения всех операций с плавающей запятой. Применение таких типов не дает какого-либо измеримого эффекта, за исключением получения более точного результата. Однако в выражениях вида x \* y / z, в которых в результате умножения получается значение, выходящее за рамки диапазона типа double, но при последующем делении может быть получен временный результат, принадлежащий диапазону double, вычисление выражения с более высокой точностью позволяет получить определенный результат вместо бесконечности.

### Тип decimal

Тип decimal представляет собой 128-битовый тип данных, ориентированный на применение в финансовых и денежных вычислениях. Тип decimal представляет значения в диапазоне от 1,0 x 10−28 до 7,9 x 1028 (приблизительно) с 28–29 значащими цифрами.

Тип decimal определяет конечный набор значений вида (–1)sx c x10-e, где параметр s может принимать значения 0 или 1, коэффициент c принадлежит диапазону 0 ≤ c < 296, а значение степени e принадлежит диапазону 0 ≤ e ≤ 28. Тип decimal не поддерживает нули и бесконечности со знаками, а также нечисловые значения. Тип decimal представляется в виде 96-разрядного целого числа, умноженного на определенную степень десяти. Значения типа decimal с абсолютной величиной менее 1.0m имеют точность не более 28 десятичных разрядов. Значения типа decimal с абсолютной величиной не менее 1.0m имеют точность 28 или 29 разрядов. В отличие от типов данных float и double, с помощью типа decimal возможно точное представление десятичных дробей, например 0,1. В представлениях float и double такие числа зачастую представляют собой бесконечные дроби, что увеличивает вероятность возникновения ошибок округления.

Если один из операндов бинарного оператора имеет тип decimal, второй операнд должен иметь целочисленный тип или тип decimal. Перед выполнением операции целочисленный операнд (при его наличии) преобразуется к типу decimal.

В результате выполнения операции получается результат типа decimal. Результат определяется путем вычисления точного результата (с сохранением масштаба, определенного для каждого оператора) и последующего округления в соответствии с разрядностью представления. Результат округляется до ближайшего допустимого значения. Если результат одинаково близок к двум допустимым значениям, округление выполняется до значения, в самом младшем разряде которого содержится четное число (так называемое «банковское округление»). Нулевой результат всегда имеет знак 0 и масштаб 0.

Если в результате выполнения арифметической операции с использованием типа decimal получается значение с абсолютной величиной меньшей или равной 5 ×10-29, операция возвращает нуль. Если в результате выполнения арифметической операции с использованием типа decimal получается значение, превышающее максимально допустимое для типа decimal, возникает исключение System.OverflowException.

Тип decimal обеспечивает более высокую точность, однако обладает меньшим диапазоном по сравнению с типами с плавающей запятой. Таким образом, при преобразовании значений, имеющих тип с плавающей запятой, в значение типа decimal возможно возникновение исключений переполнения. При преобразовании из типа decimal в типы с плавающей запятой возможно снижение точности результата. В связи с этим не существует неявного преобразования между типами с плавающей запятой и типом decimal. Чтобы использовать операнды с плавающей запятой и операнды типа decimal в одном выражении, необходимо выполнить явное приведение их значений.

### Тип bool

Тип bool представляет логические величины. Тип bool поддерживает два возможных значения: true и false.

Стандартных преобразований между типом bool и другими типами не существует. В частности, тип bool содержит конечное число значений и является отдельным от целых типов. Не допускается использование значений типа bool вместо целочисленных значений и наоборот.

В языках C и C++ нулевые целочисленные значения или значения с плавающей запятой, а также пустые указатели могут быть преобразованы в логическое значение false. Ненулевые целочисленные значения или значения с плавающей запятой и непустые указатели могут быть преобразованы в логическое значение true. В C# такие преобразования выполняются посредством явного сравнения целочисленного значения или значения с плавающей запятой с нулем, а также посредством явного сравнения ссылки на объект со значением null.

### Перечисляемые типы

Перечисляемые типы содержат конечное число значений, представляющих собой именованные константы. Для каждого перечисляемого типа существует базовый тип, в качестве которого должен используется один из следующих типов: byte, sbyte, short, ushort, int, uint, long и ulong. Набор значений перечисляемого типа соответствует набору значений базового типа. Возможные значения перечисляемого типа не ограничиваются значениями именованных констант. Определение перечисляемых типов выполняется посредством объявлений перечисления (§14.1).

### Обнуляемые типы

Обнуляемые типы могут представлять все значения базового типа и дополнительное пустое значение. Обнуляемый тип обозначается как T?, где T — базовый тип. Данный синтаксис представляет собой сокращенное обозначение типа System.Nullable<T>. Эти синтаксические формы являются взаимозаменяемыми.

Напротив, необнуляемый тип значений представляет собой тип значений, отличный от System.Nullable<T> или его сокращенной формы T? (для любого типа T)), плюс любой параметр типа, который должен иметь необнуляемый тип значений (то есть любой параметр типа с ограничением struct). Тип System.Nullable<T> определяет ограничение типа значений для типа T ( (§10.1.5). Это означает, что в качестве базового для обнуляемого типа может использоваться любой необнуляемый тип значений. Обнуляемые или ссылочные типы не могут использоваться в качестве базовых для обнуляемого типа. Например, типы int?? и string? не являются допустимыми.

Экземпляр обнуляемого типа T? обладает двумя открытыми свойствами, доступными только для чтения:

* Свойство HasValue типа bool.
* Свойство Value типа T.

Экземпляр, свойство HasValue которого имеет значение true, считается непустым. Свойство Value непустого экземпляра содержит возвращаемое значение.

Экземпляр, свойство HasValue которого имеет значение false, считается пустым. Пустой экземпляр имеет неопределенное значение. Попытка чтения свойства Value пустого экземпляра приводит к исключению System.InvalidOperationException. Процесс доступа к свойству Value обнуляемого экземпляра называется развертыванием.

Помимо конструктора по умолчанию, для каждого обнуляемого типа T? предусмотрен открытый конструктор, принимающий один аргумент типа T. При вызове конструктора со значением x типа T в виде

new T?(x)

создается непустой экземпляр типа T?, свойству Value которого присваивается значение x. Процесс создания непустого экземпляра обнуляемого типа с использованием заданного значения называется свертыванием.

Доступны неявные преобразования литерала null к типу T? (§6.1.5) и типа T к T? (§6.1.4).

## Ссылочные типы

Предусмотрены следующие виды ссылочных типов: тип класса, тип интерфейса, тип массива и тип делегата.

reference-type:  
class-type  
interface-type  
array-type  
delegate-type

class-type:  
type-name  
object  
dynamic  
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

Значение ссылочного типа представляет собой ссылку на экземпляр типа (объект). Особое значение null совместимо со всеми ссылочными типами и обозначает отсутствие экземпляра.

### Типы классов

Тип класса определяет структуру данных, которая содержит данные-члены (константы и поля), функции-члены (методы, свойства, события, индексаторы, операторы, конструкторы экземпляров, деструкторы и статические конструкторы), а также вложенные типы. Типы классов поддерживают механизм наследования, который позволяет создавать производные классы, расширяющие функциональные возможности базового класса. Экземпляры типов классов создаются с помощью выражений создания объекта (§7.6.10.1).

Дополнительные сведения о типах классов см. в разделе §10.

В языке C# некоторые предопределенные типы классов имеют особое значение (см. таблицу ниже).

|  |  |
| --- | --- |
| **Тип класса** | **Описание** |
| System.Object | Первичный базовый класс для всех типов. См. §4.2.2. |
| System.String | Строковый тип языка C#. См. §4.2.4. |
| System.ValueType | Базовый класс для всех типов значений. См. §4.1.1. |
| System.Enum | Базовый класс для всех перечисляемых типов. См. §14. |
| System.Array | Базовый класс для всех типов массивов. См. §12. |
| System.Delegate | Базовый класс для всех типов делегатов. См. §15. |
| System.Exception | Базовый класс для всех типов исключений. См. §16. |

### Тип объекта

Тип класса object является первичным базовым классом для всех типов. В C# все типы прямо или косвенно наследуются от типа класса object.

Ключевое слово object — это всего лишь псевдоним предопределенного класса System.Object.

### Динамический тип

Тип dynamic, как и тип object, может ссылаться на любой объект. Если операторы применяются к выражениям типа dynamic, их разрешение откладывается на время выполнения программы. Таким образом, если оператор не может быть легально применен к ссылочному объекту, во время компиляции ошибок не возникает. Вместо этого вызывается исключение при сбое разрешения оператора во время выполнения программы.

Более подробно динамический тип описывается в §4.7, а динамическая привязка в §7.2.2.

### Строковый тип

Тип string представляет собой запечатанный тип класса, наследуемый непосредственно от класса object. Экземпляры класса string представляют собой строки, состоящие из символов Юникода.

Значения типа string могут быть записаны в виде строковых литералов (§2.4.4.5).

Ключевое слово string представляет собой псевдоним предопределенного класса System.String.

### Типы интерфейсов

В интерфейсе определяется контракт. Класс или структура, в которых реализуется этот интерфейс, должны соблюдать условия данного контракта. Интерфейс может наследовать от нескольких базовых интерфейсов, а в классе или структуре может быть реализовано несколько интерфейсов.

Дополнительные сведения о типах интерфейса см. в разделе §13.

### Типы массивов

Массив представляет собой структуру данных, содержащую нуль или более переменных, доступ к которым осуществляется с помощью вычисляемых индексов. Все переменные, содержащиеся в массиве, которые также называются элементами массива, имеют одинаковый тип, который называется типом элементов массива.

Дополнительные сведения о типах массивов см. в разделе §12.

### Типы делегатов

Делегат представляет собой структуру данных, содержащую ссылки на один или несколько методов. Для методов экземпляров делегат также содержит ссылки на соответствующие экземпляры объектов.

Ближайшим аналогом делегата в C и C++ является указатель на функцию. Различие между ними заключается в том, что указатель на функцию может содержать ссылки только на статические функции, тогда как делегат может содержать ссылки как на статические методы, так и на методы экземпляров. В последнем случае в делегате хранится не только ссылка на точку входа метода, но и ссылка на экземпляр объекта, для которого осуществляется вызов метода.

Дополнительные сведения о типах делегатов см. в разделе §15.

## Упаковка и распаковка

Понятие упаковки и распаковки является ключевым в системе типов C#. С помощью данных операций осуществляется связь между типами значений и ссылочными типами за счет возможности преобразования любого значения типа значений к типу object и обратно. С помощью упаковки и распаковки обеспечивается унифицированное представление системы типов, в котором значение любого типа может обрабатываться как объект.

### Преобразования упаковки

Преобразование упаковки обеспечивает неявное преобразование типа значений в ссылочный тип. Предусмотрены следующие преобразования упаковки:

* из любого типа значений к типу object;
* из любого типа значений к типу System.ValueType;
* из любого необнуляемого типа значений к любому типу интерфейса, реализованному с помощью типа значений;
* из любого обнуляемого типа к любому типу интерфейса, реализованному с помощью базового типа для обнуляемого типа;
* из любого перечисляемого типа к типу System.Enum;
* из любого обнуляемого типа, базовым для которого является перечисляемый тип, к типу System.Enum.

Обратите внимание, что если неявное преобразование из параметра типа во время выполнения включает в себя преобразование из типа значений к ссылочному типу, оно будет выполнено как преобразование упаковки (§6.1.10).

Упаковка значения включает в себя выделение экземпляра объекта и копирование значения необнуляемого типа значений в указанный экземпляр.

Если выполняется упаковка значения null (свойство HasValue имеет значение false) обнуляемого типа, то при этом создается пустая ссылка. В противном случае возвращается результат развертывания и упаковки базового значения.

Чтобы более наглядно представить процесс упаковки значения необнуляемого типа значений, предположим существование универсального класса упаковки, поведение которого могло бы определяться следующим объявлением:

sealed class Box<T>: System.ValueType  
{  
 T value;

public Box(T t) {  
 value = t;  
 }  
}

Упаковка значения v типа T будет включать в себя выполнение выражения new Box<T>(v) с последующим возвратом результата в виде значения типа object. Таким образом, операторы

int i = 123;  
object box = i;

будут соответствовать операторам

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

В действительности описанный выше класс упаковки Box<T> не существует, а динамический тип упакованного значения не является типом класса. Упакованное значение типа T имеет динамический тип T. При проверке динамического типа с помощью оператора is возможно использование ссылки на тип T. Например,

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

на консоль будет выведена строка "Box contains an int".

Преобразование упаковки подразумевает создание копии упаковываемого значения. В этом заключается отличие от преобразования ссылочного типа к типу object, при котором значение продолжает хранить ссылку на тот же экземпляр и считается производным значением типа object более низкого уровня. Например, при объявлении

struct Point  
{  
 public int x, y;

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

в результате выполнения операторов

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

на консоль будет выведено значение 10, поскольку при выполнении операции неявной упаковки (осуществляется при присвоении значения p свойству box) происходит копирование значения p. Если вместо Point объявить class, будет выведено значение 20, поскольку в этом случае p и box будут ссылаться на один экземпляр.

### Преобразования распаковки

Преобразование распаковки обеспечивает явное преобразование ссылочного типа к типу значений. Предусмотрены следующие преобразования распаковки:

* Из типа object к любому типу значений.
* Из типа System.ValueType к любому типу значений.
* Из любого типа интерфейса к любому необнуляемому типу значений, реализующему тип интерфейса.
* Из любого типа интерфейса к любому обнуляемому типу, базовый тип которого реализует тип интерфейса.
* Из типа System.Enum к любому перечисляемому типу.
* Из типа System.Enum к любому обнуляемому типу, базовым для которого является перечисляемый тип.

Обратите внимание, что если явное преобразование из параметра типа во время выполнения включает в себя преобразование из ссылочного типа к типу значений, оно будет выполнено как преобразование распаковки (§6.2.6).

При выполнении операции распаковки для необнуляемого типа значений сначала проверяется, является ли экземпляр объекта упакованным значением указанного необнуляемого типа значений. После этого выполняется копирование значения из экземпляра.

Если исходный операнд для операции распаковки в обнуляемый тип имеет значение null, результатом операции будет пустое значение обнуляемого типа. В противном случае возвращается свернутый результат распаковки экземпляра объекта в тип, являющийся базовым для обнуляемого типа.

Если рассматривать предполагаемый класс упаковки, описанный в предыдущем разделе, преобразование распаковки объекта box к типу значений T будет включать в себя выполнение выражения ((Box<T>)box).value. Таким образом, операторы

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

будут соответствовать операторам

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

Для успешного выполнения преобразования распаковки в указанный необнуляемый тип значений значение исходного операнда должно содержать ссылку на упакованное значение этого необнуляемого типа значений. Если исходный операнд имеет значение null, возникает исключение System.NullReferenceException. Если исходный операнд содержит ссылку на несовместимый объект, порождается исключение System.InvalidCastException.

Для успешного выполнения операции распаковки в заданный обнуляемый тип исходный операнд должен иметь значение null или содержать ссылку на упакованное значение необнуляемого типа значений, являющегося базовым для обнуляемого типа. Если исходный операнд содержит ссылку на несовместимый объект, порождается исключение System.InvalidCastException.

## Сформированные типы

Объявление универсального типа определяет несвязанный универсальный тип, который используется в качестве шаблона для формирования различных типов посредством применения аргументов типа. Аргументы типа записываются с помощью угловых скобок (< и >) непосредственно после имени универсального типа. Тип, содержащий как минимум один аргумент типа, называется сформированным типом. Сформированный тип может использоваться в большинстве конструкций языка, в которых используется имя типа. Несвязанный универсальный тип может использоваться только в выражении typeof (§7.6.11).

Сформированные типы также могут использоваться в выражениях в качестве простых имен (§7.6.2) или при доступе к членам (§7.6.4).

При вычислении пространства имен или имени типа рассматриваются только универсальные типы, содержащие допустимое число параметров. Таким образом, возможно обозначение типов, имеющих различное число параметров, с помощью одного идентификатора. Это полезно при одновременном использовании в программе универсальных классов и классов, не являющихся таковыми:

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  
 }  
}

Имя типа должно определять сформированный тип, даже если в нем прямо не заданы параметры типа. Это может произойти, если тип является вложенным в объявлении универсального класса, а тип экземпляра в содержащем объявлении неявно используется для поиска имен (§10.3.8.6):

class Outer<T>  
{  
 public class Inner {...}

public Inner i; // Type of i is Outer<T>.Inner  
}

В небезопасном коде не допускается использование сформированного типа в качестве неуправляемого типа (§18.2).

### Аргументы типа

Каждый аргумент в списке аргументов типа представляет собой тип.

type-argument-list:  
< type-arguments >

type-arguments:  
type-argument  
type-arguments , type-argument

type-argument:  
type

В небезопасном коде (§18) аргумент типа не может являться типом указателя. Каждый аргумент типа должен удовлетворять любым ограничениям, накладываемым на соответствующий параметр типа (§10.1.5).

### Открытые и закрытые типы

Все типы можно подразделить на открытые и закрытые типы. Открытый тип — это тип, в котором используются параметры типа. В частности:

* Параметр типа определяет открытый тип.
* Массив имеет открытый тип только в том случае, если тип элементов массива является открытым.
* Сформированный тип является открытым только в том случае, если один или несколько его аргументов типа имеют открытый тип. Вложенный сформированный тип является открытым только в том случае, если один или несколько аргументов типа для него или для содержащих его типов имеют открытый тип.

Все остальные типы являются закрытыми.

Во время выполнения код, находящийся внутри объявления универсального типа, выполняется в контексте закрытого сформированного типа, который создается посредством применения аргументов типа к универсальному объявлению. Каждый параметр типа в универсальном типе связан с определенным типом времени выполнения. Обработка всех операторов и выражений во время выполнения всегда происходит с использованием закрытых типов. Открытые типы используются только при обработке во время компиляции.

Каждый закрытый сформированный тип обладает собственным набором статических переменных, которые недоступны для совместного использования с любым другим закрытым сформированным типом. Поскольку во время выполнения не существуют открытые типы, статические переменные, связанные с ними, отсутствуют. Два закрытых сформированных типа имеют одинаковый тип, если они созданы на базе одного несвязанного универсального типа и их соответствующие аргументы типа имеют одинаковый тип.

### Связанные и несвязанные типы

Термин несвязанный тип обозначает не являющийся универсальным тип или несвязанный универсальный тип. Термин связанный тип обозначает не являющийся универсальным тип или сформированный тип.

Несвязанный тип ссылается на сущность, описанную в объявлении типа. Несвязанный универсальный тип сам по себе не является типом и не может использоваться в качестве типа переменной, аргумента или возвращаемого значения, а также в качестве базового типа. Несвязанный универсальный тип может использоваться только в выражениях typeof (§7.6.11).

### Соблюдение ограничений

При каждом использовании сформированного типа или универсального метода выполняется проверка предоставленных аргументов типа на предмет соответствия ограничениям, накладываемым на параметры типа при объявлении универсального типа или метода (§10.1.5). Для каждого предложения where выполняется проверка аргумента типа A, соответствующего параметру именованного типа, следующим образом:

* Если ограничение представляет собой тип класса, тип интерфейса или параметр типа, определяется тип C, представляющий это ограничение. Предоставленные аргументы типа замещают любые параметры типа, указанные в ограничении. Соответствие ограничению соблюдается только в том случае, если тип A можно преобразовать к типу C с помощью любого из следующих преобразований:
* Преобразование идентификации (§6.1.1).
* Неявное преобразование ссылочного типа (§6.1.6)
* Преобразование упаковки (§6.1.7), если тип A представляет собой необнуляемый тип значений.
* Неявное преобразование ссылочного типа, упаковки или параметра типа из параметра типа A в C.
* Если ограничение является ограничением ссылочного типа (class), тип A должен соответствовать одному из следующих ограничений:
* Тип A является типом интерфейса, класса, делегата или массива. Обратите внимание, что типы System.ValueType и System.Enum являются ссылочными типами, которые соответствуют этому ограничению.
* Тип A представляет собой параметр типа, являющегося ссылочным типом (§10.1.5).
* Если ограничение является ограничением типа значений (struct), тип A должен соответствовать одному из следующих ограничений:
* Тип A является типом структуры или перечисляемым типом и не является обнуляемым типом. Обратите внимание, что типы System.ValueType и System.Enum являются ссылочными типами, которые не соответствуют этому ограничению.
* Тип A представляет собой параметр типа, для которого определено ограничение типа значений (§10.1.5).
* Если ограничение является ограничением конструктора new(), тип A не может представлять класс abstract и должен включать не содержащий параметров открытый конструктор. Соответствие этому ограничению обеспечивается в том случае, если верно одно из следующих утверждений:
* Тип A является типом значений, поскольку все типы значений содержат открытый конструктор по умолчанию (§4.1.2).
* Тип A представляет собой параметр типа, для которого определено ограничение конструктора (§10.1.5).
* Тип A представляет собой параметр типа, для которого определено ограничение типа значений (§10.1.5).
* Тип A представляет собой класс, отличный от abstract, который содержит явно объявленный конструктор public, не имеющий параметров.
* Тип A не является классом abstract и содержит конструктор по умолчанию (§10.11.4).

Если для указанных аргументов типа не соблюдаются одно или несколько ограничений параметров типа, возникает ошибка времени компиляции.

Поскольку наследование параметров типа не поддерживается, наследование ограничений также невозможно. В приведенном ниже примере требуется задать ограничение для параметра типа T типа D. При этом параметр T должен соответствовать ограничению, заданному в базовом классе B<T>. Напротив, для класса E не требуется задавать ограничение, поскольку List<T> реализует IEnumerable для любого T.

class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}

## Параметры типа

Параметр типа представляет собой идентификатор, определяющий тип значений или ссылочный тип, с которым параметр связан во время выполнения.

type-parameter:  
identifier

Поскольку экземпляр параметра типа может быть создан с несколькими различными аргументами фактического типа, операции и ограничения, присущие параметрам типа, несколько отличаются от других типов. Основные отличия:

* Не допускается прямое использование параметра типа для объявления базового класса (§10.2.4) или интерфейса (§13.1.3).
* Правила поиска членов для параметров типа определяются применяемыми к ним ограничениями (при наличии таковых). Дополнительные сведения об ограничениях см. в разделе §7.4.
* Доступные преобразования для параметров типа определяются применяемыми к ним ограничениями (при наличии таковых). Дополнительные сведения о преобразованиях см. в разделах §6.1.10 и §6.2.6.
* Литерал null не может быть преобразован к типу, задаваемому параметром типа, за исключением случаев, когда параметр является ссылочным типом (§6.1.10). Вместо этого можно использовать выражение default (§7.6.13). Кроме того, если для параметра типа не предусмотрено ограничение типа значений, можно сравнить значение, которое имеет тип, задаваемый параметром типа, со значением null с помощью операторов == и != (§7.10.6).
* Выражение new (§7.6.10.1) может использоваться только с параметром типа, для которого предусмотрено ограничение конструктора или ограничение типа значений (§10.1.5).
* Параметр типа не может использоваться в атрибутах.
* Параметр типа не может использоваться при доступе к члену (§7.6.4) или в имени типа (§3.8) для определения статического члена или вложенного типа.
* В небезопасном коде не допускается использование параметра типа в качестве неуправляемого типа (§18.2).

Являясь типом, параметр типа представляет собой конструкцию, существующую исключительно во время компиляции. Во время выполнения каждый параметр типа связан с типом времени выполнения, заданным с помощью аргумента типа в объявлении универсального типа. Таким образом, переменная, объявленная с помощью параметра типа, во время выполнения будет иметь закрытый сформированный тип (§4.4.2). Во время выполнения для операторов и выражений, содержащих параметры типа, используются фактические типы, предоставленные в качестве аргументов типа для этих параметров.

## Типы дерева выражений

Деревья выражений обеспечивают представление лямбда-выражений в виде структур данных вместо исполняемого кода. Дерево выражений представляет собой значение типа дерева выражения вида System.Linq.Expressions.Expression<D>, где D — любой тип делегата. Далее в этой спецификации такие типы будут обозначаться с помощью сокращенной формы Expression<D>.

Если для лямбда-выражения существует преобразование к типу делегата D, для нее также существует и преобразование к типу дерева выражений Expression<D>. В результате преобразования лямбда-выражения к типу делегата создается делегат, содержащий ссылки на исполняемый код этого лямбда-выражения. При преобразовании лямбда-выражения к типу дерева выражений создается представление этого лямбда-выражения в виде дерева выражений.

Дерево выражений является эффективным представлением данных лямбда-выражения в памяти, обеспечивающим прозрачное и явное представление структуры этого выражения.

Аналогично типу делегата D, тип Expression<D> обладает типами параметров и возвращаемых значений, которые совпадают с параметрами типа D.

В приведенном ниже примере лямбда-выражение представляется как в виде исполняемого кода, так и в виде дерева выражений. Поскольку существует преобразование к Func<int,int>, также существует и преобразование к Expression<Func<int,int>>:

Func<int,int> del = x => x + 1; // Code

Expression<Func<int,int>> exp = x => x + 1; // Data

В результате этих присвоений делегат del содержит ссылку на метод, который возвращает x + 1, а дерево выражений exp содержит ссылку на структуру данных, описывающую выражение x => x + 1.

Точное определение универсального типа Expression<D>, а также точные правила построения дерева выражений при преобразовании лямбда-выражения к типу дерева выражений описываются в соответствующей документации.

Следует обратить внимание на следующие моменты:

* Не все лямбда-выражения можно преобразовывать в деревья выражений. Например, не поддерживается представление лямбда-выражений, содержащих тела оператора и выражения присваивания. В этих случаях преобразование по-прежнему существует, однако вызывает ошибку во время компиляции. Подробное описание этих исключений см. в разделе §6.5.
* Тип Expression<D> содержит метод экземпляра Compile, порождающий делегат типа D:

Func<int,int> del2 = exp.Compile();

При вызове этого делегата выполняется код, представленный деревом выражений. Таким образом, учитывая приведенные выше определения, del и del2 являются эквивалентными, то есть выполнение приведенных ниже операторов имеет одинаковый эффект:

int i1 = del(1);

int i2 = del2(1);

После выполнения этого кода переменным i1 и i2 присваивается значение 2.

## Динамический тип

Тип dynamic в C# имеет особое значение. Он необходим для обеспечения возможности динамической привязки, которая описана подробно в §7.2.2.

Тип dynamic считается идентичным типу object за исключением следующих случаев:

* если операции и выражения типа dynamic могут быть использованы в динамической привязке (§7.2.2);
* вывод типа (§7.5.2) предпочтет тип dynamic типу object при наличии обоих вариантов.

Ввиду этой равнозначности, верно, что:

* Существует неявное тождественное преобразование между типами object и dynamic, а также между сформированными типами, которые не изменяются при замене типа dynamic на тип object.
* Явные и неявные преобразования из типа object и к этому типу также применяются для преобразования к типу dynamic и из этого типа.
* Сигнатуры метода, которые являются теми же, когда заменяют тип dynamic на тип object, считаются теми же сигнатурами.

Тип dynamic не отличается от типа object во время выполнения.

Выражение типа dynamic указывает на то, что это динамическое выражение.

# Переменные

Переменные указывают места хранения данных. Каждая переменная имеет тип, который определяет, какие значения могут храниться в этой переменной. В языке C# все данные должны относиться к определенному типу, компилятор C# обеспечивает соответствие типа значений, хранящихся в переменных. Значение переменной можно изменить путем присваивания или при помощи операторов ++ и ‑‑.

Получить значение переменной можно только в том случае, если она является определенно присвоенной (§5.3).

Как рассматривается в следующих разделах, переменные подразделяются на переменные с начальным значением и переменные без начального значения. Переменная с начальным значением имеет должным образом определенное исходное значение и всегда считается определенно присвоенной. У переменной без начального значения исходное значение отсутствует. Чтобы переменная без начального значения считалась определенно присвоенной в конкретном месте, необходимо присвоить этой переменной значение в каждом возможном пути выполнения, ведущем к указанному месту.

## Категории переменных

В C# определено семь категорий переменных: статические переменные, переменные экземпляра, элементы массива, параметры по значению, параметры по ссылке, выходные параметры и локальные переменные. Эти категории рассматриваются в следующих ниже разделах.

В этом примере

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++;  
 }  
}

Переменная x является статической переменной, y — переменной экземпляра, v[0] — элементом массива, a — параметром по значению, b — параметром по ссылке, c — выходным параметром, а i — локальной переменной.

### Статические переменные

Поле, объявленное с модификатором static, называется статической переменной. Статическая переменная создается перед выполнением статического конструктора (§10.12) для содержащегося в ней типа и прекращает свое существование при удалении домена связанного приложения.

Начальным значением статической переменной является значение, установленное по умолчанию для соответствующего типа переменных (§5.2).

Для целей проверки определенного присваивания статическая переменная считается переменной с начальным значением.

### Переменные экземпляра

Поле, объявленное без модификатора static, называется переменной экземпляра.

#### Переменные экземпляра в классах

Переменная экземпляра класса создается при создании нового экземпляра этого класса и удаляется в том случае, если отсутствуют ссылки на этот экземпляр и выполнен деструктор экземпляра (при его наличии).

Начальным значением переменной экземпляра класса является значение, установленное по умолчанию для данного типа переменных (§5.2).

Для целей проверки определенного присваивания переменная экземпляра класса считается переменной с начальным значением.

#### Переменные экземпляра в структурах

Переменная экземпляра в структуре имеет такое же время жизни, как и переменная структуры, которой она принадлежит. Другими словами, переменные экземпляра в структуре создаются и удаляются при создании и удалении соответствующей структуры.

Состояние наличия начального значения переменной экземпляра в структуре соответствует состоянию переменной структуры, которой она принадлежит. Другими словами, если переменная структуры считается имеющей начальное значение, это справедливо и для принадлежащих ей переменных экземпляра. Если же переменная структуры считается не имеющей начального значения, ее переменные экземпляра также лишены начального значения.

### Элементы массива

Элементы массива создаются при создании экземпляра массива и удаляются при отсутствии ссылок на этот экземпляр массива.

Начальное значение каждого из элементов массива равно значению по умолчанию для элементов массива данного типа (§5.2).

Для целей проверки определенного присваивания элемент массива считается переменной с начальным значением.

### Параметры по значению

Параметр, объявленный без модификаторов ref и out, является параметром по значению.

Параметр по значению создается при вызове соответствующей функции-члена (метода, конструктора экземпляра, метода доступа или оператора) или анонимной функции, к которой относится данный параметр, и инициализируется по значению аргумента, заданного при вызове функции. Параметр по значению, как правило, удаляется при возвращении значения функцией-членом или анонимной функцией. Тем не менее, если параметр по значению перехвачен анонимной функцией (§7.15), его время жизни увеличивается на срок, пока делегат или дерево выражений, созданные из этой анонимной функции, доступны для сбора мусора.

Для целей проверки определенного присваивания параметр по значению считается переменной с начальным значением.

### Параметры ссылок

Параметр, объявленный с модификатором ref, является параметром по ссылке.

Для параметра ссылки не создается новое место хранения. Параметр по ссылке указывает место хранения переменной, заданной в качестве аргумента при вызове функции-члена или анонимной функции. Таким образом, значение параметра по ссылке всегда совпадает со значением базовой переменной.

В отношении параметров по ссылке применяются указанные ниже правила определенного присваивания. Обратите внимание на разные правила для выходных параметров, описанные в §5.1.6.

* Переменная должна быть определенно присвоена (§5.3), чтобы ее можно было передать как параметр по ссылке при вызове функции-члена или делегата.
* В функции-члене или анонимной функции параметр по ссылки считается имеющим начальное значение.

В методе экземпляра или методе доступа экземпляра с типом структуры ключевое слово this имеет точно такое же поведение, как параметр по ссылке с типом структуры (§7.6.7).

### Выходные параметры

Параметр, объявленный с модификатором out, является выходным параметром.

При использовании выходного параметра не создается новое место хранения. Выходной параметр указывает место хранения переменной, заданной в качестве аргумента при вызове функции-члена или делегата. Таким образом, значение выходного параметра всегда совпадает со значением базовой переменной.

В отношении выходных параметров применяются указанные ниже правила определенного присваивания. Обратите внимание на разные правила для выходных параметров, описанные в §5.1.5.

* Для передачи переменной как выходного параметра при вызове функции-члена или делегата определенно присваивать переменную необязательно.
* После нормального завершения вызова функции-члена или делегата каждая переменная, переданная как выходной параметр, считается имеющей значение в этом пути выполнения.
* В функции-члене или анонимной функции выходной параметр считается не имеющим начального значения.
* Каждый выходной параметр функции-члена или анонимной функции должен быть определенно присвоен (§5.3) перед нормальным завершением выполнения такой функции-члена или анонимной функции.

В конструкторе экземпляра с типом структуры ключевое слово this имеет точно такое же поведение, как выходной параметр с типом структуры (§7.6.7).

### Локальные переменные

Локальная переменная объявляется с использованием объявления локальной переменной, которое может находиться в блоке, операторе for, операторе switch или операторе using, а также при помощи оператора foreach или конкретной конструкции catch в операторе try.

Время жизни локальной переменной соответствует части выполнения программы, в которой для нее гарантированно резервируется место. Это время жизни длится по меньшей мере с момента входа в блок, операторы for, switch, using, foreach или конкретную конструкцию catch, с которой связана данная переменная, до завершения выполнения этого блока, операторов for, switch, using, foreach или конкретной конструкции catch любым из способов. (Вход во вложенный блок или вызов метода приостанавливает, но не завершает выполнение текущих блока, операторов for, switch, using, foreach или конкретной конструкции catch.) Если локальная переменная перехватывается анонимной функцией (§7.15.5.1), ее время жизни увеличивается по крайней мере на период времени, в течение которого делегат или дерево выражений, созданные в этой анонимной функции, а также другие объекты, ссылающиеся на перехваченную переменную, станут доступны для сбора мусора.

Если вход в родительский блок, оператор for, switch, using, foreach или конкретную конструкцию catch выполняется рекурсивно, каждый раз создается новый экземпляр локальной переменной, при этом каждый раз вычисляется инициализатор локальной переменной (при его наличии).

Локальная переменная, созданная с использованием объявления локальной переменной, не инициализируется автоматически и поэтому не имеет значения по умолчанию. Для целей проверки определенного присваивания локальная переменная, созданная с использованием объявления локальной переменной, считается не имеющей начального значения. Объявление локальной переменной может содержать инициализатор локальной переменной. В этом случае переменная считается определенно присвоенной только после инициализирующего выражения (§5.3.3.4).

В области действия локальной переменной, созданной с использованием объявления локальной переменной, ссылка на эту локальную переменную из фрагмента программы, предшествующего в тексте объявлению локальной переменной, приведет к ошибке компиляции. Если объявление локальной переменной выполнено неявно (§8.5.1), ссылка на эту переменную в ее объявлении локальной переменной также вызовет ошибку.

Локальная переменная, созданная с использованием оператора foreach или конкретной конструкции catch, считается определенно присвоенной по всей области действия.

Фактическое время жизни локальной переменной зависит от реализации. Например, компилятор может статическим образом определить, что локальная переменная в блоке используется лишь в небольшой части этого блока. В такого анализа компилятор может создать код, в котором время жизни места хранения переменной будет меньше, чем время жизни содержащего ее блока.

Место хранения, на которое ссылается локальная ссылочная переменная, освобождается без учета времени жизни этой ссылочной переменной (§3.9).

## Значения по умолчанию

Указанные ниже категории переменных автоматически инициализируются соответствующими значениями по умолчанию.

* Статические переменные.
* Переменные экземпляра класса.
* Элементы массива.

Значение по умолчанию для переменной зависит от типа переменной и определяется следующим образом:

* Для переменной с типом значений, значение по умолчанию равно значению, вычисленному конструктором по умолчанию для типа значений (§4.1.2).
* Для переменной со ссылочным типом значением по умолчанию является null.

При инициализации значениями по умолчанию, как правило, диспетчер памяти или сборщик мусора перед выделением области памяти инициализирует все ее биты нулями. По этой причине пустую ссылку удобно представить как область памяти, все биты которой равны нулю.

## Определенное присваивание

В конкретной позиции исполняемого кода функции-члена переменная считается определенно присвоенной, если в результате отдельного статического анализа потока (§5.3.3) компилятор выявляет, что эта переменная была автоматически инициализирована или стала целевым объектом по меньшей мере одного присваивания. Другими словами, действуют следующие правила определенного присваивания:

* Переменная с начальным значением (§5.3.1) всегда считается определенно присвоенной.
* Переменная без начального значения (§5.3.2) считается определенно присвоенной в указанном расположении, если все возможные пути выполнения, ведущие в это расположение, содержат по меньшей мере одну из следующих операций:
* Простое присваивание (§7.17.1), в котором переменная является левым операндом.
* Выражение вызова (§7.6.5) или выражение создания объекта (§7.6.10.1), передающее переменную в качестве выходного параметра.
* Для локальной переменной — объявление локальной переменной (§8.5.1), включающее инициализатор переменной.

Формальная спецификация, определяющая изложенные выше правила, описана в разделах §5.3.1, §5.3.2 и §5.3.3.

Определенное присваивание для переменных экземпляра переменной с типом структуры отслеживаются как по отдельности, так и в совокупности. В дополнение к изложенным выше правилам в отношении переменных с типом структуры и их переменных экземпляра применяются следующие правила:

* Переменная экземпляра считается определенно присвоенной, если содержащая ее переменная с типом структуры считается определенно присвоенной.
* Переменная с типом структуры считается определенно присвоенной, если каждая из ее переменных экземпляра считается определенно присвоенной.

Определенное присваивание является обязательным в следующих контекстах:

* Переменная должна быть определенно присвоена в каждом месте, где извлекается ее значение. Это обеспечивает отсутствие неопределенных значений. Включение переменной в выражение считается извлечением значения этой переменной за исключением следующих случаев:
* переменная является левым операндом выражения простого присваивания,
* переменная передается в качестве выходного параметра, или
* переменная является переменной с типом структуры и указана как левый операнд метода доступа к члену.
* Переменная должна быть определенно присвоена в каждой точке, где она передается как параметр по ссылке. Это гарантирует, что вызываемая функция-член сможет использовать этот параметр по ссылке с начальным значением.
* Все выходные параметры функции-члена должны быть определенно присвоены в каждой точке, где выполняется возврат из функции-члена (при помощи оператора return или в результате достижения конца тела функции-члена). Это гарантирует, что функции-члены не будут возвращать в выходных параметрах неопределенные значения, что позволяет компилятору обработать вызов функции-члена, принимающей переменную в качестве выходного параметра, как выполнение присваивания в отношении этой переменной.
* Переменная this конструктора экземпляра переменной с типом структуры должна быть определенно присвоена в каждой точке, где выполняется возврат из этого конструктора экземпляра.

### Переменные с начальным значением

К переменным с начальным значением относятся следующие категории переменных:

* Статические переменные.
* Переменные экземпляра класса.
* Переменные экземпляра переменных с типом struct с начальным значением.
* Элементы массива.
* Параметры по значению.
* Параметры по ссылке.
* Переменные, объявленные в предложении catch или операторе foreach.

### Переменные без начального значения

К переменным без начального значения относятся следующие категории переменных:

* Переменные экземпляра переменных с типом struct без начального значения.
* Выходные параметры, включая переменную this конструкторов экземпляра переменных с типом структуры.
* Локальные переменные, за исключением объявленных в предложении catch или операторе foreach.

### Точные правила для выявления определенного присваивания

Чтобы удостовериться, что каждая из используемых переменных является определенно присвоенной, компилятор должен использовать процесс, аналогичный рассмотренному в этом разделе.

Компилятор обрабатывает тело каждой функции-члена, имеющей одну или несколько переменных без начального значения. Для каждой переменной v без начального значения компилятор определяет состояние определенного присваивания для v в каждой из следующих точек функции-члена:

* В начале каждого оператора
* В конечной точке каждого оператора (§8.1)
* В каждой ветке, где управление передается в другой оператор или в конечную точку оператора
* В начале каждого выражения
* В конце каждого выражения

Состояние определенного присваивания переменной v может иметь одно из следующих значений:

* С определенным присваиванием. Указывает, что во всех возможных потоках управления в эту точку переменной v было присвоено значение.
* Без определенного присваивания. Состояние переменной без определенного присваивания в конце выражения с типом bool может (но не должно) иметь одно из следующих значений дополнительного состояния:
* Определенно присвоенная после выполнения выражения с итогом true. Это состояние указывает, что переменная v является определенно присвоенной, если в результате выполнения логического выражения было получено значение true, однако может не являться присвоенной, если в результате выполнения логического выражения было получено значение false.
* Определенно присвоенная после выполнения выражения с итогом false. Это состояние указывает, что переменная v является определенно присвоенной, если в результате выполнения логического выражения было получено значение false, однако может не являться присвоенной, если в результате выполнения логического выражения было получено значение true.

Определение состояния переменной v в каждой из точек управляется указанными ниже правилами.

#### Общие правила для операторов

* Переменная v не является определенно присвоенной в начале тела функции-члена.
* Переменная v является определенно присвоенной в начале каждой недостижимого оператора.
* Состояние определенного присваивания переменной v в начале каждого другого оператора определяется путем проверки состояния определенного присваивания переменной v во всех ветвлениях потоков управления, ведущих к началу этого оператора. Только в том случае, если переменная v является определенно присвоенной во всех таких ветвлениях потоков управления, переменная v считается определенно присвоенной в начале оператора. Набор возможных ветвлений потоков управления определяется точно так же, как и при проверке достижимости операторов (§8.1).
* Состояние определенного присвоения переменной v в конечной точке блока или оператора checked, unchecked, if, while, do, for, foreach, lock, using или switch определяется путем проверки состояния определенного присвоения переменной v во всех ветвях потоков управления, ведущих к конечной точке этого оператора. Если переменная v является определенно присвоенной во всех ветвях потоков управления, переменная v считается определенно присвоенной в конечной точке оператора. В противном случае переменная v не является определенно присвоенной в конечной точке оператора. Набор возможных ветвлений потоков управления определяется точно так же, как и при проверке достижимости операторов (§8.1).

#### Операторы блока, операторы checked и unchecked

Состояние определенного присваивания переменной v в ветви потока управления, ведущей к первому оператору в списке операторов блока (либо к конечной точке блока, если список операторов пуст) соответствует состоянию определенного присваивания переменной v перед блоком, оператором checked или unchecked.

#### Операторы-выражения

Для операции stmt, состоящего из выражений expr:

* переменная v имеет такое же состояние определенного присваивания к началу выражения expr, как и к началу оператора stmt;
* Если переменная v является определенно присвоенной к концу выражения expr, она считается определенно присвоенной в конечной точке оператора stmt; в противном случае она не является определенно присвоенной в конечной точке оператора stmt.

#### Операторы объявления

* Если оператор stmt является оператором объявления без инициализаторов, переменная v имеет такое же состояние определенного присваивания в конце оператора stmt, как и в начале оператора stmt.
* Если оператор stmt является оператором объявления с инициализаторами, состояние определенного присваивания для переменной v определяется таким образом, словно оператор stmt является списком операторов с отдельным оператором присваивания для каждого объявления с инициализатором (в порядке объявления).

#### Операторы If

Для операции stmt оператора if в форме:

if ( expr ) then-stmt else else-stmt

* переменная v имеет такое же состояние определенного присваивания к началу выражения expr, как и к началу оператора stmt;
* если переменная v является определенно присвоенной на конец выполнения выражения expr, она является определенно присвоенной в ветви потока управления до операции then-stmt, а также или в else-stmt, или в конечной точке операции stmt, если условие else отсутствует;
* если переменная v имеет состояние "определенно присвоенное в результате выполнения выражения с итогом true" в конце выражения expr, она считается определенно присвоенной в ветви потока управления до операции then-stmt и не является определенно присвоенной в ветви потока управления или в else-stmt, или в конечной точке операции stmt, если условие else отсутствует;
* если переменная v имеет состояние "определенно присвоенная после выполнения выражения с результатом false" в конце выражения expr, она является определенно присвоенной в ветви else-stmt и не является определенно присвоенной в ветви then-stmt. Эта переменная считается определенно присвоенной в конечной точке ветви stmt только в том случае, если она является определенно присвоенной в конечной точке then-stmt;
* в обратном случае переменная v не считается определенно присвоенной в ветви потока управления до операции then-stmt или else-stmt либо до конечной точки ветви stmt, если условие else отсутствует.

#### Операторы switch

В операторе выражения stmt оператора switch с управляющим выражением expr:

* состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию переменной v в начале операции stmt;
* состояние определенного присваивания переменной v в ветви потока управления до списка достижимых операторов блока switch соответствует состоянию определенного присваивания переменной v в конце выражения expr.

#### Операторы while

Для операции stmt оператора while в форме:

while ( expr ) while-body

* переменная v имеет такое же состояние определенного присваивания к началу выражения expr, как и к началу оператора stmt;
* если переменная v является определенно присвоенной в конце выражения expr, она является определенно присвоенной в ветви потока управления до операции while-body и до конечной точки операции stmt;
* если переменная v имеет состояние "определенно присвоенное в результате выполнения выражения с итогом true" в конце выражения expr, она является определенно присвоенной в ветви потока управления до операции while-body, однако не является определенно присвоенной в конечной точке операции stmt;
* если переменная v имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false" в конце выражения expr, она является определенно присвоенной в ветви потока управления до конечной точки операции stmt, однако не является определенно присвоенной в ветви потока управления до операции while-body.

#### Операторы do

Для операции stmt оператора do в форме:

do do-body while ( expr ) ;

* переменная v имеет такое же состояние определенного присваивания в ветви потока управления с начала операции stmt до операции do-body, как и в начале операции stmt.
* переменная v имеет такое же состояние определенного присваивания в начале выражения expr, как и в конечной точке операции do-body.
* если переменная v является определенно присвоенной в конечной точке expr, она является определенно присвоенной в ветви потока управления до конечной точки операции stmt.
* если переменная v имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false" в конце выражения expr, она является определенно присвоенной в ветви потока управления до конечной точки stmt.

#### Операторы for

Проверка определенного присваивания для оператора for в форме:

for ( for-initializer ; for-condition ; for-iterator ) embedded-statement

выполняется для следующей формы оператора:

{  
 for-initializer ;  
 while ( for-condition ) {  
 embedded-statement ;  
 for-iterator ;  
 }  
}

Если условие for в операторе for опущено, оценка определенного присваивания выполняется таким образом, словно условие for в приведенной выше развернутой записи оператора было заменено значением true.

#### Операторы break, continue и goto

Состояние определенного присваивания переменной v в ветви потока управления, вызванной оператором break, continue или goto, соответствует состоянию определенного присваивания переменной v в начале оператора.

#### Операторы throw

Для операции stmt оператора в форме

throw expr ;

Состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию определенного присваивания переменной v в начале оператора stmt.

#### Операторы return

Для операции stmt оператора в форме

return expr ;

* Состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию определенного присваивания переменной v в начале оператора stmt.
* Если переменная v является выходным параметром, она должна быть определенно присвоена в одном из следующих случаев:
* после выражения expr;
* либо в конце блока finally конструкции try-finally или try-catch-finally, содержащей оператор return.

Для stmt оператора в форме:

return ;

* Если переменная v является выходным параметром, она должна быть определенно присвоена в одном из следующих случаев:
* перед оператором stmt;
* либо в конце блока finally конструкции try-finally или try-catch-finally, содержащей оператор return.

#### Операторы try-catch

Для stmt оператора в форме:

try try-block  
catch(...) catch-block-1  
...  
catch(...) catch-block-n

* Состояние определенного присваивания переменной v в начале try-block соответствует состоянию определенного присваивания переменной v в начале операции stmt.
* Состояние определенного присваивания переменной v в начале catch-block-i (для любой переменной i) соответствует состоянию определенного присваивания переменной v в начале операции stmt.
* Переменная v в конечной точке операции stmt считается определенно присвоенной только в том случае, если переменная v определенно присвоена в конечной точке try-block и каждого catch-block-i (для каждой переменной i от 1 до n).

#### Операторы try-finally

Для операции stmt оператора try в форме:

try try-block finally finally-block

* Состояние определенного присваивания переменной v в начале try-block соответствует состоянию определенного присваивания переменной v в начале операции stmt.
* Состояние определенного присваивания переменной v в начале finally-block соответствует состоянию определенного присваивания переменной v в начале операции stmt.
* Переменная v в конечной точке операции stmt считается определенно присвоенной только при соблюдении по меньшей мере одного из следующих условий:
* переменная v является определенно присвоенной в конечной точке try-block;
* переменная v является определенно присвоенной в конечной точке finally-block.

При переключении потока управления (например, в операторе goto), которое начинается в try-block и завершается за пределами try-block, переменная v также считается определенно присвоенной в такой ветви потока управления, если переменная v является определенно присвоенной в конечной точке finally-block. (Это не единственная возможность — если переменная v является определенно присвоенной по другой причине при таком переключении потока управления, она все еще считается определенно присвоенной.)

#### Операторы try-catch-finally

Анализ определенного присвоения для оператора try-catch-finally в виде:

try try-block   
catch(...) catch-block-1  
...  
catch(...) catch-block-n  
finally finally-block

выполняется с допущением, что оператор try-finally содержит оператор try-catch:

try {  
 try try-block   
 catch(...) catch-block-1  
 ...  
 catch(...) catch-block-n  
}  
finally finally-block

В следующем примере демонстрируется, как разные блоки оператора try (§8.10) влияют на определенное присваивание.

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  
  
 }  
}

#### Операторы foreach

Для операции stmt оператора foreach в форме:

foreach ( type identifier in expr ) embedded-statement

* состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию переменной v в начале операции stmt;
* состояние определенного присваивания переменной v при переключении потока управления на внедренный оператор или в конечную точку операции stmt соответствует состоянию переменной v в конце выражения expr.

#### Операторы using

Для операции stmt оператора using в форме:

using ( resource-acquisition ) embedded-statement

* состояние определенного присваивания переменной v в начале операции resource-acquisition соответствует состоянию переменной v в начале операции stmt.
* Состояние определенного присваивания переменной v при переключении потока управления на внедренный оператор соответствует состоянию переменной v в конце операции resource-acquisition.

#### Операторы lock

Для операции stmt оператора lock в форме:

lock ( expr ) embedded-statement

* состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию переменной v в начале операции stmt;
* Состояние определенного присваивания переменной v при переключении потока управления на внедренный оператор соответствует состоянию переменной v в конце выражения expr.

#### Операторы yield

Для операции stmt оператора yield return в форме:

yield return expr ;

* состояние определенного присваивания переменной v в начале выражения expr соответствует состоянию переменной v в начале операции stmt;
* состояние определенного присваивания переменной v в конце операции stmt соответствует состоянию переменной v в конце выражения expr.

Оператор yield break не влияет на состояние определенного присвоения.

#### Общие правила для простых выражений

Указанное ниже правило применяется к следующим видам выражений: литералы (§7.6.1), простые имена (§7.6.2), выражения доступа к членам (§7.6.4), неиндексированные выражения базового доступа (§7.6.8), выражения typeof (§7.6.11) и выражения значений по умолчанию (§7.6.13).

* Состояние определенного присваивания переменной v в конце такого выражения соответствует состоянию определенного присваивания переменной v в начале этого выражения.

#### Общие правила для выражений с внедренными выражениями

Указанные ниже правила применяются к следующим видам выражений: выражения со скобками (§7.6.3), выражения доступа к элементам (§7.6.6), выражения базового доступа с индексацией (§7.6.8), выражения приращения и уменьшения (§7.6.9, §7.7.5), выражения приведения (§7.7.6), унарные выражения +, - ~, \*, бинарные выражения +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ (§7.8, §7.9, §7.10, §7.11), составные выражения присваивания (§7.17.2), выражения checked и unchecked (§7.6.12), а также выражения создания массивов и делегатов (§7.6.10).

Каждое из этих выражений имеет одно или несколько вложенных выражений, которые подлежат безусловному вычислению в фиксированном порядке. Например, в бинарном операторе % сначала вычисляется левая сторона оператора, а затем — правая сторона. В операции индексирования вычисляется индексированное выражение, а затем вычисляется каждое из выражений индекса в порядке слева направо. Выражение expr, которое имеет вложенные выражения expr1, expr2, ..., exprn, вычисляется в следующем порядке:

* Состояние определенного присваивания переменной v в начале выражения expr1 соответствует состоянию определенного присваивания в начале выражения expr.
* Состояние определенного присваивания переменной v в начале выражения expri (i больше единицы) соответствует состоянию определенного присваивания в конце выражения expri-1.
* Состояние определенного присваивания переменной v в конце выражения expr соответствует состоянию определенного присваивания в конце выражения exprn.

#### Выражения вызова и выражения создания объекта

Для выражения вызова expr в форме:

primary-expression ( arg1 , arg2 , … , argn )

или выражения создания объекта в форме

new type ( arg1 , arg2 , … , argn )

* для выражения вызова состояние определенного присваивания переменной v перед первичным выражением соответствует состоянию переменной v перед выражением expr;
* для выражения вызова состояние определенного присваивания переменной v перед arg1 соответствует состоянию переменной v после первичного выражения;
* для выражения создания объекта состояние определенного присваивания переменной v перед arg1 соответствует состоянию переменной v перед выражением expr;
* для каждого аргумента argi состояние определенного присваивания переменной v после argi определяется обычными правилами выражений с игнорированием модификаторов ref и out;
* для каждого аргумента argi при i больше единицы состояние определенного присваивания переменной v перед argi соответствует состоянию переменной v после argi-1;
* если переменная v передается в качестве аргумента out (например, аргумента в виде out v) в любом из аргументов, переменная v после выражения expr является определенно присвоенной. В противном случае состояние переменной v после выражения expr соответствует состоянию переменной v после argn;
* для инициализаторов массива (§7.6.10.4), инициализаторов объектов (§7.6.10.2), инициализаторов коллекций (§7.6.10.3) и инициализаторов анонимных объектов (§7.6.10.6) состояние определенного присваивания определяется подстановкой, в контексте которой определяются эти конструкции.

#### Простые выражения присваивания

Для выражения expr в форме w = expr-rhs:

* состояние определенного присваивания переменной v перед выражением expr-rhs соответствует состоянию определенного присваивания переменной v перед выражением expr.
* если переменная w является той же самой переменной, что и переменная v, переменная v после выражения expr является определенно присвоенной. В противном случае состояние определенного присваивания переменной v после выражения expr соответствует состоянию определенного присваивания переменной v после выражения expr-rhs.

#### Выражения &&

Для выражения expr в форме expr-first && expr-second:

* состояние определенного присваивания переменной v перед выражением expr-first соответствует состоянию определенного присваивания переменной v перед выражением expr;
* переменная v перед выражением expr-second является определенно присвоенной, если состояние переменной v после выражения expr-first является определенно присвоенным или "определенно присвоенным в результате выполнения выражения с итогом true". В обратном случае она не является определенно присвоенной;
* состояние определенного присваивания переменной v после выражения expr определяется следующими правилами:
* если выражение expr-first является константным выражением, имеющим значение false, состояние определенного присваивания переменной v после выражения expr соответствует состоянию определенного присваивания переменной v после выражения expr-first;
* в противном случае, если переменная v после выражения expr-first является определенно присвоенной, переменная v после выражения expr является определенно присвоенной;
* в противном случае, если состояние переменной v после выражения expr-second является определенно присвоенным, а состояние переменной v после выражения expr-first соответствует состоянию "определенно присвоенная в результате выполнения выражения с итогом false", переменная v после выражения expr является определенно присвоенной;
* в противном случае, если переменная v после выражения expr-second является определенно присвоенной или "определенно присвоенной в результате выполнения выражения с итогом true", переменная v после выражения expr является "определенно присвоенной в результате выполнения выражения с итогом true";
* в противном случае, если переменная v после выражения expr-first имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false", а после выражения expr-second переменная v имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false", то после выражения expr переменная v имеет состояние "определенно присвоенная в результате выражения с итогом false";
* в противном случае переменная v после выражения expr не является определенно присвоенной.

В этом примере

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  
 }  
}

переменная i считается определенно присвоенной только в одном из внедренных операторов оператора if. В операторе if в методе F переменная i является определенно присвоенной в первом внедренном операторе, поскольку выполнение выражения (i = y) всегда предшествует выполнению этого внедренного оператора. Наоборот, переменная i не является определенно присвоенной во втором внедренном операторе, поскольку проверка условия x >= 0 может завершиться с итогом false, в результате чего для переменной i не будет выполнено присваивание.

#### Выражения ||

Для выражения expr в форме expr-first || expr-second:

* состояние определенного присваивания переменной v перед выражением expr-first соответствует состоянию определенного присваивания переменной v перед выражением expr;
* переменная v перед выражением expr-second является определенно присвоенной, если состояние переменной v после выражения expr-first является определенно присвоенным или "определенно присвоенным в результате выполнения выражения с итогом false". В обратном случае она не является определенно присвоенной;
* состояние определенного присваивания переменной v после выражения expr определяется следующими факторами:
* если выражение expr-first является константным выражением, имеющим значение true, состояние определенного присваивания переменной v после выражения expr соответствует состоянию определенного присваивания переменной v после выражения expr-first;
* в противном случае, если переменная v после выражения expr-first является определенно присвоенной, переменная v после выражения expr является определенно присвоенной;
* в противном случае, если состояние переменной v после выражения expr-second является определенно присвоенным, а состояние переменной v после выражения expr-first соответствует состоянию "определенно присвоенная в результате выполнения выражения с итогом true", переменная v после выражения expr является определенно присвоенной;
* в обратном случае, если переменная v после выражения expr-second является определенно присвоенной или "определенно присвоенной в результате выполнения выражения с итогом false", переменная v после выражения expr является "определенно присвоенной в результате выполнения выражения с итогом false";
* в обратном случае, если переменная v после выражения expr-first имеет состояние "определенно присвоенная в результате выполнения выражения с итогом true", а после выражения expr-second имеет переменная v состояние "определенно присвоенная в результате выполнения выражения с итогом true", переменная v после выражения expr имеет состояние "определенно присвоенная в результате выражения с итогом true";
* в противном случае переменная v после выражения expr не является определенно присвоенной.

В этом примере

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  
 }  
}

переменная i считается определенно присвоенной только в одном из внедренных операторов оператора if. В операторе if в методе G переменная i является определенно присвоенной во втором внедренном операторе, поскольку выполнение выражения (i = y) всегда предшествует выполнению этого внедренного оператора. Наоборот, переменная i не является определенно присвоенной в первом внедренном операторе, поскольку проверка условия x >= 0 может завершиться с итогом true, в результате чего переменной i не будет выполнено присваивание.

#### ! выражения

Для выражения expr в форме ! expr-operand:

* состояние определенного присваивания переменной v перед выражением expr-operand соответствует состоянию определенного присваивания переменной v перед выражением expr.
* состояние определенного присваивания переменной v после выражения expr определяется следующими правилами:
* если переменная v после выражения expr-operand является определенно присвоенной, переменная v после выражения expr также является определенно присвоенной;
* если переменная v после выражения expr-operand не является определенно присвоенной, переменная v после выражения expr также не является определенно присвоенной;
* если переменная v после выражения expr-operand имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false", переменная v после выражения expr имеет состояние "определенно присвоенная в результате выполнения выражения с итогом true";
* если переменная v после выражения expr-operand имеет состояние "определенно присвоенная в результате выполнения выражения с итогом true", переменная v после выражения expr имеет состояние "определенно присвоенная в результате выполнения выражения с итогом false".

#### Выражения ??

Для выражения expr в форме expr-first?? expr-second:

* состояние определенного присваивания переменной v перед выражением expr-first соответствует состоянию определенного присваивания переменной v перед выражением expr;
* состояние определенного присваивания переменной v перед выражением expr-second соответствует состоянию определенного присваивания переменной v после выражения expr-first.
* состояние определенного присваивания переменной v после выражения expr определяется следующими факторами:
* если выражение expr-first является константным выражением (§7.19), имеющим значение NULL, состояние переменной v после выражения expr соответствует состоянию переменной v после выражения expr-second;
* В обратном случае состояние переменной v после выражения expr соответствует состоянию определенного присваивания переменной v после выражения expr-first.

#### Выражения ?:

Для выражения expr в форме expr-cond ? expr-true : expr-false:

* состояние определенного присваивания переменной v перед выражением expr-cond соответствует состоянию переменной v перед выражением expr;
* Переменная v перед выражением expr-true считается определенно присвоенной только при соблюдении одного из следующих условий:
* выражение expr-cond является константным выражением со значением false;
* переменная v после выражения expr-cond является определенно присвоенной или "определенно присвоенной в результате выполнения выражения с результатом true";
* Переменная v перед выражением expr-false считается определенно присвоенной только при соблюдении одного из следующих условий:
* expr-cond является константным выражением, имеющим значение true
* переменная v после выражения expr-cond является определенно присвоенной или "определенно присвоенной в результате выполнения выражения с результатом false";
* состояние определенного присваивания переменной v после выражения expr определяется следующими правилами:
* если выражение expr-cond является константным выражением (§7.19), имеющим значение true, состояние переменной v после выражения expr соответствует состоянию переменной v после выражения expr-true;
* в обратном случае, если выражение expr-cond является константным выражением (§7.19), имеющим значение false, состояние переменной v после выражения expr соответствует состоянию переменной v после выражения expr-false;
* в обратном случае, если переменная v является определенно присвоенной после выражения expr-true, а также переменная v является определенной присвоенной после выражения expr-false, переменная v после выражения expr также является определенно присвоенной;
* в противном случае переменная v после выражения expr не является определенно присвоенной.

#### Анонимные функции

Для лямбда-выражений или выражений анонимного метода expr с телом body, содержащим блок или выражение:

* Состояние определенного присваивания внешней переменной v перед телом body соответствует состоянию переменной v перед выражением expr. Таким образом, состояние определенного присваивания внешних переменных наследуется из контекста анонимной функции.
* Состояние определенного присваивания внешней переменной v после выражения expr соответствует состоянию переменной v перед выражением expr.

Пример:

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);  
}

приводит к возникновению ошибки времени компиляции, поскольку переменная max не является определенно присвоенной при объявлении анонимной функции. Пример:

delegate void D();

void F() {  
 int n;  
 D d = () => { n = 1; };

d();

// Error, n is not definitely assigned  
 Console.WriteLine(n);  
}

также приводит к возникновению ошибки времени компилирования, поскольку присваивание для переменной n в анонимной функции не влияет на состояние определенного присваивания переменной n вне анонимной функции.

## Ссылочные переменные

Ссылка на переменную представляет собой выражение, которое классифицируется как переменная. Ссылка на переменную указывает место хранения, к которому можно получить доступ как для извлечения текущего значения, так и для сохранения нового значения.

variable-reference:  
expression

В C и C++ ссылка на переменную называется lvalue.

## Атомарность ссылок на переменные

К атомарным действиям относятся чтение и запись следующих типов данных: bool, char, byte, sbyte, short, ushort, uint, int, float и ссылочных типов. В дополнение к этому атомарными действиями являются чтение и запись типов перечислений с базовым типом в приведенном выше списке. Чтение и запись других типов, включая long, ulong, double и decimal, а также пользовательских типов, не обязательно являются атомарными действиями. Если речь не идет о предназначенных для этой цели библиотечных функциях, гарантия атомарного чтения, изменения и записи, например в случае приращения или уменьшения, отсутствует.

# Преобразования

С помощью преобразований выражения могут обрабатываться как выражения определенного типа. Преобразования позволяют обрабатывать выражения заданного типа как имеющие другой тип, а выражения без типа – как выражения определенного типа. Преобразования бывают явными и неявными, что определяет необходимость использования явного приведения. Например, преобразование из типа int к типу long является неявным, поэтому выражения типа int могут неявно обрабатываться как тип long. Противоположное преобразование — из типа long к типу int — является явным, поэтому в этом случае требуется явное приведение.

int a = 123;  
long b = a; // implicit conversion from int to long  
int c = (int) b; // explicit conversion from long to int

Некоторые преобразования определяются языком. В программе также можно определить собственные преобразования (§6.4).

## Неявные преобразования

К неявным преобразованиям относятся следующие:

* Преобразования идентификатора
* Неявные числовые преобразования
* Неявные преобразования перечисляемых типов.
* Неявные преобразования обнуляемых типов.
* Преобразования литерала NULL.
* Неявные преобразования ссылочных типов
* Преобразования упаковки.
* Неявные преобразования динамических типов
* Неявные преобразования выражений констант
* Пользовательские неявные преобразования
* Преобразования анонимных функций
* Преобразования группы методов

Неявные преобразования могут происходить в разнообразных ситуациях, включая вызовы членов функций (§7.5.4), выражения приведения (§7.7.6) и присваивания (§7.17).

Предопределенные неявные выражения всегда выполняются успешно и никогда не порождают исключений. Корректно разработанные пользовательские преобразования также должны отвечать этим требованиям.

С точки зрения преобразования, типы object и dynamic считаются эквивалентными.

Однако динамические преобразования (§6.1.8 and §6.2.6) применяются только к выражениям типа dynamic (§4.7).

### Преобразование идентификатора

Преобразование идентификатора обеспечивает приведение типа к такому же типу. Это преобразование используется только для того, чтобы задать возможность преобразования к конкретному типу сущности, уже имеющей этот тип.

Поскольку типы object и dynamic считаются эквивалентными, существует тождественное преобразование между типами object и dynamic, а также между сформированными типами, которые не изменяются при замене всех экземпляров типа dynamic на тип object.

### Неявные числовые преобразования

Существуют следующие неявные преобразования числовых типов:

* Из sbyte в short, int, long, float, double или decimal.
* Из byte в short, ushort, int, uint, long, ulong, float, double или decimal.
* Из short в int, long, float, double или decimal.
* Из ushort в int, uint, long, ulong, float, double или decimal.
* Из int в long, float, double или decimal.
* Из типа uint в тип long, ulong, float, double или decimal.
* Из long в float, double или decimal.
* Из типа ulong в тип float, double или decimal.
* Из char в ushort, int, uint, long, ulong, float, double или decimal.
* Из типа float в тип double.

Преобразования из int, uint, long или ulong в float и из long или ulong в double могут привести к потере точности, но никогда не приведут к потере величины. Другие неявные преобразования числовых типов никогда не приводят к потере данных.

Из-за отсутствия неявных преобразований в тип char значения других целочисленных типов не преобразуются в тип char автоматически.

### Неявные преобразования перечисляемых типов

Неявное преобразование перечисляемых типов позволяет преобразовать десятичный целочисленный литерал 0 в любой перечисляемый тип, а также в любой обнуляемый тип с базовым перечисляемым типом. В последнем случае выполняется преобразование к базовому перечисляемому типу и последующее свертывание результата (§4.1.10).

### Неявные преобразования обнуляемых типов

Предопределенные неявные преобразования для необнуляемых типов значений могут также использоваться и с обнуляемыми формами таких типов. Для каждого предопределенного неявного преобразования идентификатора или числового типа, которое выполняет преобразование из необнуляемого типа значений S к аналогичному типу T, существуют следующие неявные преобразования обнуляемых типов:

* Неявное преобразование из типа S? в тип T?.
* Неявное преобразование из типа S в тип T?.

Вычисление неявного преобразования обнуляемого типа, основанного на преобразовании из S к T, осуществляется следующим образом:

* Если выполняется преобразование обнуляемых типов из S? в T?:
* Если исходное значение равно NULL (значение свойства HasValue равно false), результатом является значение NULL типа T?.
* В противном случае преобразование вычисляется посредством развертывания из S? в S, преобразования из S к T и последующего свертывания (§4.1.10) из T к T?.
* Если выполняется преобразование обнуляемых типов из S в T?, преобразование вычисляется как базовое преобразование из S в T с последующим свертыванием из типа T в тип T?.

### Преобразования литерала NULL.

Существует неявное преобразование литерала null в любой обнуляемый тип. В результате преобразования получается значение NULL (§4.1.10) заданного обнуляемого типа.

### Неявные преобразования ссылочных типов

Существуют следующие неявные преобразования ссылочных типов:

* Из любого ссылочного типа в типы object и dynamic.
* Из любого типа класса S к любому типу класса T (где S является производным от T).
* Из любого типа класса S к любому типу интерфейса T при условии, что S реализует T.
* Из любого типа интерфейса S к любому типу интерфейса T (где S является производным от T).
* Из типа массива S с элементами типа SE в тип массива T с элементами типа TE при условии выполнения следующих требований:
* S и T различаются только по типу элементов. Другими словами, S и T имеют одинаковое число измерений.
* SE и TE являются ссылочными типами.
* Существует неявное преобразование ссылочных типов из SE в TE.
* Из любого типа массива в тип System.Array и реализуемые этим типом интерфейсы.
* Из одномерного типа массива S[] к System.Collections.Generic.IList<T> и его базовым интерфейсам, если существует неявное преобразование идентификатора или ссылочного типа из S к T.
* Из любого типа делегата в тип System.Delegate и реализуемые этим типом интерфейсы.
* Из любого литерала NULL к любому ссылочному типу.
* Из любого ссылочного типа в ссылочный тип T, если для него предусмотрено неявное тождественное преобразование или преобразование ссылочных типов из ссылочного типа T0 и для типа T0 предусмотрено тождественное преобразование в тип T.
* Из любого ссылочного типа в тип интерфейса или делегата T, если для него предусмотрено неявное тождественное преобразование или преобразование ссылочных типов в тип интерфейса или делегата T0 и тип T0 является вариантно-преобразуемым (§13.1.3.2) в тип T.
* Неявные преобразования, включающие параметры типа, которые имеют ссылочный тип. Дополнительные сведения о неявных преобразованиях, включающих параметры типа, см. в §6.1.10.

Неявные преобразования ссылочных типов всегда выполняются успешно и не требуют проверки во время выполнения.

Неявные или явные преобразования ссылочных типов никогда не изменяют ссылочный идентификатор преобразуемого объекта. Другими словами, преобразование ссылочного типа может изменить тип ссылки, однако никогда не изменяет тип или значение объекта, на который указывает ссылка.

### Преобразования упаковки

Преобразование упаковки обеспечивает неявное преобразование типа значений в ссылочный тип. Преобразование упаковки возможно из любого необнуляемого типа значения в типы object и dynamic, в тип System.ValueType, а также в любой тип интерфейса, реализуемый необнуляемым типом значения. Кроме того, перечисляемый тип может быть преобразован в тип System.Enum.

Преобразование упаковки из обнуляемого типа к ссылочному типу существует только в том случае, если существует преобразование упаковки из базового необнуляемого типа значений к ссылочному типу.

Для типа значения возможно преобразование упаковки в тип интерфейса I, если для него предусмотрено преобразование упаковки в тип интерфейса I0 и для типа I0 предусмотрено тождественное преобразование в тип I.

Для типа значения возможно преобразование упаковки в тип интерфейса I, если для него предусмотрено преобразование упаковки в тип интерфейса или делегата I0 и тип I0 является вариантно-преобразуемым (§13.1.3.2) в тип I.

Упаковка значения необнуляемого типа значений включает в себя выделение экземпляра объекта и копирование этого значения типа значений в указанный экземпляр. Структура может быть упакована в тип System.ValueType, поскольку он является базовым классом для всех структур (§11.3.2).

Упаковка значения обнуляемого типа осуществляется следующим образом:

* Если исходное значение равно NULL (значение свойства HasValue равно false), результатом является пустая ссылка целевого типа.
* В противном случае результатом является ссылка на упакованное значение типа T, полученное в результате развертывания и упаковки исходного значения.

Дополнительные сведения о преобразованиях упаковки см. в §4.3.1.

### Неявные преобразования динамических типов

Существует неявное преобразование динамических типов из выражения типа dynamic в любой тип T. Преобразование является динамически привязанным (§7.2.2) — это означает, что во время выполнения будет выполняться поиск неявного преобразования типа выражения времени выполнения в тип T. Если преобразование не обнаружено, во время выполнения возникает исключение.

Обратите внимание, что это неявное преобразование противоречит совету, данному в §6.1, о том, что неявное преобразование не должно вызывать исключений. Однако по сути это не столько преобразование, сколько поискпреобразования, вызывающего исключение. Риск возникновения исключений во время выполнения является неотъемлемым при использовании динамической привязки. Если динамическая привязка преобразования не требуется, можно сначала преобразовать выражение в тип object, а затем в требуемый тип.

Следующий пример иллюстрирует неявное динамическое преобразование:

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

В присвоениях объектам s2 и i используются неявные динамические преобразования, в которых привязка операций откладывается до времени выполнения. Во время выполнения выполняется поиск неявных преобразований из типа объекта времени выполнения d — string — в целевой тип. Удается найти преобразование в тип string, но не в тип int.

### Неявные преобразования выражений констант

Неявные преобразования выражений констант обеспечивают следующие преобразования:

* Преобразование константного выражения (§7.19) типа int в тип sbyte, byte, short, ushort, uint или ulong возможно, если значение константного выражения принадлежит диапазону целевого типа.
* Преобразование константного выражения типа long к типу ulong, если значение константного выражения не является отрицательным.

### Неявные преобразования, включающие параметры типа

Существуют следующие неявные преобразования для заданного параметра типа T:

* Из T к эффективному базовому классу C, из T к любому базовому для C классу, а также из T к любому интерфейсу, реализованному классом C. Если во время выполнения T является типом значения, преобразование выполняется как преобразование упаковки. Иначе преобразование выполняется как неявное преобразование ссылочного типа или преобразование идентификации.
* Из T к типу интерфейса I, принадлежащему эффективному набору интерфейсов T, а также из T к любому базовому интерфейсу I. Если во время выполнения T является типом значения, преобразование выполняется как преобразование упаковки. Иначе преобразование выполняется как неявное преобразование ссылочного типа или преобразование идентификации.
* Из T в параметр типа U, если T зависит от U (§10.1.5). Если во время выполнения U является типом значения, T и U обязательно будут одним типом и преобразование не выполняется. Иначе, если T является типом значений, во время выполнения преобразование выполняется как преобразование упаковки. Иначе преобразование выполняется как неявное преобразование ссылочного типа или преобразование идентификации.
* Из литерала NULL к T, если T является ссылочным типом.
* Из типа T в ссылочный тип I, если для него предусмотрено неявное преобразование в ссылочный тип S0 и для типа S0 предусмотрено тождественное преобразование в тип S. Если во время выполнения это преобразование выполняется так же, как преобразование в тип S0.
* Из типа T в тип интерфейса I, если для него предусмотрено неявное преобразование в тип интерфейса или делегата I0 и тип I0 является вариантно-преобразуемым (§) в тип I (§13.1.3.2). Если во время выполнения T является типом значения, преобразование выполняется как преобразование упаковки. Иначе преобразование выполняется как неявное преобразование ссылочного типа или преобразование идентификации.

Если известно, что T является ссылочным типом (§10.1.5), все описанные выше преобразования классифицируются как неявные преобразования ссылочных типов (§6.1.6). Если T не является ссылочным типом, описанные выше преобразования классифицируются как преобразования упаковки (§6.1.7).

### Пользовательские неявные преобразования

Пользовательское неявное преобразование включает в себя необязательное стандартное неявное преобразование, за которым следует выполнение пользовательского оператора неявного преобразования, а затем другое необязательное стандартное неявное преобразование. Точные правила вычисления пользовательских неявных преобразований описываются в §6.4.4.

### Преобразования анонимных функций и преобразования группы методов

Анонимные функции и группы методов не имеют типа, однако могут быть неявно преобразованы к типу делегата или типу дерева выражений. Дополнительные сведения о преобразованиях анонимных функций и группы методов см. в §6.5 и §6.6 соответственно.

## Явные преобразования

К явным преобразованиям относятся следующие преобразования:

* Все неявные преобразования.
* Явные преобразования числовых типов.
* Явные преобразования перечисляемых типов.
* Явные преобразования обнуляемых типов.
* Явные преобразования ссылочных типов.
* Явные преобразования типов интерфейса.
* Преобразования распаковки.
* Неявные динамические преобразования.
* Пользовательские явные преобразования.

Явные преобразования могут произойти в выражениях приведения (§7.7.6).

Набор явных преобразований включает все неявные преобразования. Это означает, что допустимы избыточные выражения приведения.

Явные преобразования, не относящиеся к неявным, не всегда выполняются успешно, могут привести к потере данных и могут выполняться в областях типов, существенно различающихся для явной нотации.

### Явные преобразования числовых типов

Явные преобразования числовых типов предназначены для преобразования из одного числового типа к другому числовому типу, для которого не существует неявного преобразования (§6.1.2):

* Из типа sbyte в тип byte, ushort, uint, ulong или char.
* Из byte в sbyte и char.
* Из типа short в тип sbyte, byte, ushort, uint, ulong или char.
* Из типа ushort в тип sbyte, byte, short или char.
* Из типа int в тип sbyte, byte, short, ushort, uint, ulong или char.
* Из типа uint в тип sbyte, byte, short, ushort, int или char.
* Из типа long в тип sbyte, byte, short, ushort, int, uint, ulong или char.
* Из типа ulong в тип sbyte, byte, short, ushort, int, uint, long или char.
* Из типа char в тип sbyte, byte или short.
* Из float в sbyte, byte, short, ushort, int, uint, long, ulong, char или decimal.
* Из double в sbyte, byte, short, ushort, int, uint, long, ulong, char, float или decimal.
* Из decimal в sbyte, byte, short, ushort, int, uint, long, ulong, char, float или double.

Поскольку явные преобразования включают в себя все неявные и явные преобразования числовых типов, всегда возможно преобразование из одного числового типа к другому числовому типу с помощью выражения приведения (§7.7.6).

Явные преобразования числовых типов могут привести к потере данных или порождению исключений. Явные преобразования числовых типов осуществляются следующим образом:

* Для преобразований из одного целого типа в другой порядок выполнения зависит от контекста проверки переполнения (§7.6.12), в котором выполняется такое преобразование:
* В контексте checked преобразование завершается успешно, если значение исходного операнда находится в диапазоне целевого типа; если значение исходного операнда находится за пределами диапазона целевого типа создается исключение System.OverflowException.
* В контексте unchecked преобразование всегда выполняется успешно следующим образом:
* Если размер исходного типа превышает размер конечного, дополнительные самые старшие разряды исходного значения отбрасываются. Результат обрабатывается как значение конечного типа.
* Если размер исходного типа меньше размера конечного, размер исходного значения расширяется до размера конечного типа за счет добавления знака или нулей. Расширение знаком используется для исходных типов со знаком, расширение нулями — для исходных типов без знака. Результат обрабатывается как значение конечного типа.
* Если размеры исходного и конечного типов совпадают, исходное значение обрабатывается как значение конечного типа.
* Для преобразований из типа decimal к целому типу исходное значение округляется в сторону нуля до ближайшего целого значения. Результатом преобразования является полученное целое значение. Если результирующее целое значение находится вне диапазона целевого типа, создается исключение System.OverflowException.
* Для преобразований из типа float или double к целому типу порядок выполнения зависит от контекста проверки переполнения (§7.6.12), в котором выполняется такое преобразование:
* В контексте checked преобразование всегда выполняется следующим образом.
* Если значение операнда равно NaN или бесконечности, возникает исключение System.OverflowException.
* В противном случае исходный операнд округляется в сторону нуля до ближайшего целого значения. Если это целое значение принадлежит диапазону конечного типа, оно возвращается в качестве результата преобразования.
* В противном случае создается исключение System.OverflowException.
* В контексте unchecked преобразование всегда выполняется успешно следующим образом:
* Если операнд имеет значение NaN или равен бесконечности, результатом преобразования является неопределенное значение конечного типа.
* В противном случае исходный операнд округляется в сторону нуля до ближайшего целого значения. Если это целое значение принадлежит диапазону конечного типа, оно возвращается в качестве результата преобразования.
* В противном случае в качестве результата преобразования возвращается неопределенное значение конечного типа.
* При преобразовании из типа double в тип float значение double округляется до ближайшего значения float. Если значение double слишком мало для представления в качестве float, результатом становится положительный нуль или отрицательный нуль. Если значение double слишком велико для представления в качестве float, результатом становится положительная бесконечность или отрицательная бесконечность. Если значение double равно NaN, результатом также будет NaN.
* При преобразовании из типа float или double в тип decimal исходное значение преобразуется в представление decimal и при необходимости округляется до ближайшего числа после 28-го десятичного разряда (§4.1.7). Если исходное значение слишком мало для представления в качестве значения типа decimal, в качестве результата возвращается нуль. Если исходное значение равно NaN, бесконечности или слишком велико для представления в виде decimal, создается исключение System.OverflowException.
* При преобразовании из типа decimal в тип float или double значение decimal округляется до ближайшего значения double или float. Это преобразование может привести к потере точности, но никогда не порождает исключений.

### Явные преобразования перечисляемых типов

Поддерживаются следующие явные преобразования перечисляемых типов:

* Из типа sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double или decimal в любой перечисляемый тип.
* Из любого перечисляемого типа в тип sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double или decimal.
* Из любого перечисляемого типа к любому другому перечисляемому типу.

Явное преобразование между двумя перечисляемыми типами выполняется посредством рассмотрения любого из участвующих перечисляемых типов в качестве базового для этого перечисляемого типа с последующим выполнением явного или неявного числового преобразования результирующих типов. Например, если для перечисляемого типа E базовым является тип int, преобразование из E в тип byte выполняется как явное преобразование числового типа (§6.2.1) из int в byte, а преобразование из byte в E — как неявное преобразование числового типа (§6.1.2) из byte в int.

### Явные преобразования обнуляемых типов

Явные преобразования обнуляемых типов позволяют использовать явные преобразования, выполняемые для необнуляемых типов значений, с обнуляемыми формами таких типов. Для каждого предопределенного явного преобразования из необнуляемого типа значений S к необнуляемому типу значений T (§6.1.1, §6.1.2, §6.1.3, §6.2.1 и §6.2.2) существуют следующие преобразования обнуляемых типов:

* Явное преобразование из S? в T?.
* Явное преобразование из типа S в тип T?.
* Явное преобразование из типа S? в тип T.

Вычисление преобразования обнуляемого типа, основанного на преобразовании из S к T, осуществляется следующим образом:

* Если выполняется преобразование обнуляемых типов из S? в T?:
* Если исходное значение равно NULL (значение свойства HasValue равно false), результатом является значение NULL типа T?.
* В противном случае преобразование вычисляется посредством развертывания из S? в S, преобразования из S к T и последующего свертывания из T к T?.
* Если выполняется преобразование обнуляемых типов из S в T?, преобразование вычисляется как базовое преобразование из S в T с последующим свертыванием из типа T в тип T?.
* Если выполняется преобразование обнуляемых типов из S? в T, преобразование вычисляется как развертывание из S? в S с последующим базовым преобразованием из типа S в тип T.

Обратите внимание, что, если значение равно null, попытка преобразования обнуляемого значения приведет к исключению.

### Явные преобразования ссылочных типов

Поддерживаются следующие явные преобразования ссылочных типов:

* Из типов object и dynamic в любой другой ссылочный тип.
* Из любого типа класса S к любому типу класса T, если S является базовым классом для T.
* Из любого типа класса S к любому типу интерфейса T, если S не является запечатанным и S не реализует T.
* Из любого типа интерфейса S к любому типу класса T, если T не является запечатанным или T реализует S.
* Из любого типа интерфейса S к любому типу интерфейса T, если S не является производным от T.
* Из типа массива S с элементами типа SE в тип массива T с элементами типа TE при условии выполнения следующих требований:
* S и T различаются только по типу элементов. Другими словами, S и T имеют одинаковое число измерений.
* SE и TE являются ссылочными типами.
* Существует явное преобразование ссылочных типов из SE в TE.
* Из любого типа System.Array и реализуемых этим типом интерфейсов в любой тип массива.
* Из одномерного типа массива S[] к System.Collections.Generic.IList<T> и его базовым интерфейсам, если существует явное преобразование ссылочного типа из S к T.
* Из System.Collections.Generic.IList<S> и его базовых интерфейсов к одномерному типу массива T[], если существует явное преобразование идентификатора или ссылочного типа из S к T.
* Из любого типа System.Delegate и реализуемых этим типом интерфейсов в любой тип делегата.
* Из ссылочного типа в ссылочный тип T, если для него предусмотрено явное преобразование ссылочных типов в ссылочный тип T0 и для типа T0 предусмотрено тождественное преобразование в тип T.
* Из ссылочного типа в тип интерфейса или делегата T, если для него предусмотрено явное преобразование ссылочных типов в тип интерфейса или делегата T0 и тип T0 является вариантно-преобразуемым в тип T или тип T является вариантно-преобразуемым в тип T0 (§13.1.3.2).
* Из D<S1…Sn> в D<T1…Tn>, где D<X1…Xn> является универсальным типом делегата, D<S1…Sn> не является совместимым или идентичным типу D<T1…Tn> и для каждого параметра типа Xi типа D выполняются следующие условия:
* Если Xi является инвариантным, тип Si идентичен типу Ti.
* Если Xi является ковариантным, существует неявное или явное тождественное преобразование или преобразование ссылочных типов из Si в Ti.
* Если Xi является контрвариантным, типы Si and Ti идентичны или одновременно являются ссылочными типами.
* Явные преобразования, включающие параметры типа, которые имеют ссылочный тип. Дополнительные сведения о явных преобразованиях, включающих параметры типа, см. в §6.2.7.

Для явных преобразований ссылочных типов требуется проверка корректности во время выполнения.

Для успешного завершения явного преобразования ссылочных типов во время выполнения необходимо, чтобы значение исходного операнда было равно null или существовало неявное преобразование ссылочных типов (§6.1.6) или преобразование упаковки (§6.1.7), позволяющее преобразовать фактический тип объекта, на который ссылается исходный операнд, в целевой тип. В случае неудачного завершения явного преобразования ссылочных типов воздается исключение System.InvalidCastException.

Неявные или явные преобразования ссылочных типов никогда не изменяют ссылочный идентификатор преобразуемого объекта. Другими словами, преобразование ссылочного типа может изменить тип ссылки, однако никогда не изменяет тип или значение объекта, на который указывает ссылка.

### Преобразования распаковки

Преобразование распаковки обеспечивает явное преобразование ссылочного типа к типу значений. Преобразование распаковки можно выполнять из типов object, dynamic и System.ValueType в любой необнуляемый тип значения, а также из любого типа интерфейса в любой необнуляемый тип значения, реализующий этот тип интерфейса. Кроме того, можно выполнить распаковку типа System.Enum в любой перечисляемый тип.

Преобразование распаковки из ссылочного типа к обнуляемому типу существует в том случае, если существует преобразование распаковки из ссылочного типа к необнуляемому типу значений, являющемуся базовым для обнуляемого типа.

Для типа значения S возможно преобразование распаковки из типа интерфейса I, если для него предусмотрено преобразование распаковки из типа интерфейса I0 и для типа I0 предусмотрено тождественное преобразование в тип I.

Для типа значения S возможно преобразование распаковки из типа интерфейса I, если для него предусмотрено преобразование распаковки из типа интерфейса или делегата I0 и тип I0 является вариантно-преобразуемым в тип I или тип I является вариантно-преобразуемым в тип I0 (§13.1.3.2).

При выполнении операции распаковки сначала проверяется, является ли экземпляр объекта упакованным значением указанного типа значений. После этого выполняется копирование значения из экземпляра. При распаковке пустой ссылки в обнуляемый тип возвращается значение NULL обнуляемого типа. Структура может быть распакована из типа System.ValueType, поскольку он является базовым классом для всех структур (§11.3.2).

Дополнительные сведения о преобразованиях распаковки см. в §4.3.2.

### Неявные динамические преобразования.

Существует явное преобразование динамических типов из выражения типа dynamic в любой тип T. Преобразование является динамически привязанным (§7.2.2) — это означает, что во время выполнения будет выполняться поиск явного преобразования типа выражения времени выполнения в тип T. Если преобразование не обнаружено, во время выполнения возникает исключение.

Если динамическая привязка преобразования не требуется, можно сначала преобразовать выражение в тип object, а затем в требуемый тип.

Предположим, что определен следующий класс.

class C  
{  
 int i;

public C(int i) { this.i = i; }

public static explicit operator C(string s)   
 {  
 return new C(int.Parse(s));  
 }  
}

Следующий пример иллюстрирует явное динамическое преобразование:

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

Наилучшее преобразование o в C, найденное во время компиляции, — это явное преобразование ссылочных типов. Оно завершится сбоем во время выполнения, поскольку “1” в действительности не является C. Однако преобразование d в C, как явное динамическое преобразование, отложено до времени выполнения, когда обнаруживается и успешно выполняется пользовательское преобразование из типа d времени выполнения — string — в C.

### Явные преобразования, включающие параметры типа

Существуют следующие явные преобразования для заданного параметра типа T:

* Из эффективного базового для типа T класса C к T, а также из любого базового для C класса к T. Если во время выполнения T является типом значения, преобразование выполняется как преобразование распаковки. В противном случае преобразование выполняется как явное преобразование ссылочного типа или идентификатора.
* Из любого типа интерфейса к T. Если во время выполнения T является типом значения, преобразование выполняется как преобразование распаковки. В противном случае преобразование выполняется как явное преобразование ссылочного типа или идентификатора.
* Из T к любому типу интерфейса I, если уже не существует неявного преобразования из T к I. Если во время выполнения T является типом значения, преобразование выполняется как преобразование упаковки, за которым следует явное преобразование ссылочных типов. В противном случае преобразование выполняется как явное преобразование ссылочного типа или идентификатора.
* Из параметра типа U в T, если T зависит от U (§10.1.5). Если во время выполнения U является типом значения, T и U обязательно будут одним типом и преобразование не выполняется. Если в процессе выполнения T является типом значений, во время выполнения преобразование выполняется как преобразование распаковки. В противном случае преобразование выполняется как явное преобразование ссылочного типа или идентификатора.

Если известно, что T является ссылочным типом, все описанные выше преобразования классифицируются как явные преобразования ссылочных типов (§6.2.4). Если T не является ссылочным типом, описанные выше преобразования классифицируются как преобразования распаковки (§6.2.5).

Приведенные выше правила не допускают прямого явного преобразования из безусловного параметра типа к типу, не являющемуся интерфейсом, поскольку его результат может быть непредсказуем. Это связано с необходимостью избежать несогласованности и достичь более точного представления семантики таких преобразований. Например, рассмотрим следующее заявление:

class X<T>  
{  
 public static long F(T t) {  
 return (long)t; // Error   
 }  
}

Если разрешено прямое явное преобразование из t к int, можно предполагать, что выражение X<int>.F(7) возвратит 7L. Однако результат будет другим, поскольку стандартные преобразования числовых типов применяются только в том случае, если типы являются числовыми во время привязки. Чтобы семантика была понятной, этот пример необходимо записать следующим образом:

class X<T>  
{  
 public static long F(T t) {  
 return (long)(object)t; // Ok, but will only work when T is long  
 }  
}

Теперь этот код будет компилироваться, но при выполнении выражения X<int>.F(7) будет создано исключение, поскольку упакованный тип int не может быть явно преобразован в long.

### Пользовательские явные преобразования

Пользовательское явное преобразование включает в себя необязательное стандартное явное преобразование, за которым следует выполнение пользовательского оператора явного преобразования, а затем другое необязательное стандартное явное преобразование. Точные правила вычисления пользовательских явных преобразований описываются в §6.4.5.

## Стандартные преобразования

Стандартные преобразования — это предопределенные преобразования, которые могут произойти в составе пользовательских.

### Стандартные неявные преобразования

К стандартным неявным преобразованиям относятся следующие преобразования:

* Преобразования идентификатора (§6.1.1)
* Неявные преобразования числовых типов (§6.1.2).
* Неявные преобразования обнуляемых типов (§6.1.4).
* Неявные преобразования ссылочных типов (§6.1.6).
* Преобразования упаковки (§6.1.7).
* Неявные преобразования выражений констант (§6.1.8).
* Неявные преобразования, включающие параметры типа (§6.1.10).

К стандартными неявным преобразованиям не относятся пользовательские неявные преобразования.

### Стандартные явные преобразования

К стандартным явным преобразованиям относятся все стандартные неявные преобразования, а также подмножество явных преобразований, для которых существуют обратные стандартные неявные преобразования. Другими словами, если существует стандартное неявное преобразование из типа A к типу B, также существует стандартное явное преобразование из типа A к B и из типа B к A.

## Пользовательские преобразования

В C# можно дополнить предопределенные явные и неявные преобразования пользовательскими преобразованиями. Пользовательские преобразования определяются посредством объявления операторов преобразования (§10.10.3) в типах класса и структуры.

### Допустимые пользовательские преобразования

В C# допускается объявление лишь некоторых пользовательских преобразований. В частности, не допускается переопределение уже существующего явного или неявного преобразования.

Для данного исходного типа S и конечного типа T, если S или T является типом, допускающим присваивание пустой ссылки, пусть S0 и T0 ссылаются на свои основные типы, иначе S0 и T0 равны S и T соответственно. В классе или структуре разрешено объявлять преобразование от исходного типа S к конечному типу T, если только справедливо все следующее:

* S0 и T0 являются разными типами;
* либо S0, либо T0 является типом структуры или класса, где имеет место объявление этого оператора;
* ни S0, ни T0 не является типом интерфейса;
* без преобразований, определенных пользователем, не существует преобразование от S к T или от T к S.

Дополнительные сведения об ограничениях, накладываемых на пользовательские преобразования, см. в §10.10.3.

### Операторы преобразования с нулификацией

Для заданного пользовательского оператора преобразования из необнуляемого типа значений S к необнуляемому типу значений T существует оператор преобразования с нулификацией, который используется для преобразования из S? к T?. Этот оператор преобразования с нулификацией выполняет развертывание из типа S? в тип S, за которым следует пользовательское преобразование из типа S в тип T, после чего выполняется развертывание из типа T в тип T?. Исключение из этого процесса составляет тип S? со значением NULL, который преобразуется непосредственно в тип T? со значением NULL.

Оператор преобразования с нулификацией классифицируется точно так же (явным или неявным), как и соответствующий ему базовый пользовательский оператор преобразования. Термин «пользовательское преобразование» применяется как к пользовательским операторам преобразования, так и к операторам с нулификацией.

### Вычисление пользовательских преобразований

Пользовательское преобразование предназначено для преобразования значения из собственного исходного типа к другому, конечному типу. Вычисление пользовательского преобразования основывается на поиске пользовательского оператора преобразования, наиболее подходящего для заданных исходного и конечного типов. Поиск оператора осуществляется в несколько этапов:

* Поиск набора классов и структур, на основании которых будут рассматриваться пользовательские операторы преобразования. Этот набор включает в себя исходный и конечный типы, а также их базовые классы (при этом явно подразумевается, что пользовательские операторы могут быть объявлены только в классах и структурах, и типы, отличные от типов класса, не могут иметь базовых классов). На этом этапе, если исходный или конечный тип является обнуляемым типом, вместо него используется базовый тип.
* На основании этого набора типов определяется, какие пользовательские операторы и операторы преобразования с нулификацией могут быть применены. Оператор преобразования применим только в том случае, если можно выполнить стандартное преобразование (§6.3) из исходного типа к типу операнда оператора, а также стандартное преобразование из типа результата оператора к конечному типу.
* На основании набора применимых пользовательских операторов определяется, какой из операторов является однозначно наиболее подходящим. В общих чертах, наиболее подходящим оператором является тот, для которого тип операнда наиболее «близок» к исходному типу, а тип результата — к конечному типу. Пользовательские операторы преобразования являются более предпочтительными по сравнению с операторами преобразования с нулификацией. Точные правила определения наиболее подходящего пользовательского оператора преобразования описываются в последующих разделах.

После определения наиболее подходящего пользовательского оператора преобразования осуществляется фактическое выполнение пользовательского преобразования, включающее в себя три этапа:

* Сначала, если это необходимо, выполняется стандартное преобразование из исходного типа к типу операнда для пользовательского оператора преобразования или оператора с нулификацией.
* Затем вызывается пользовательский оператор преобразования или оператор с нулификацией и выполняется преобразование.
* В завершение, если это необходимо, выполняется стандартное преобразование из типа результата для пользовательского оператора преобразования или оператора с нулификацией к конечному типу.

При вычислении пользовательского преобразования никогда не используется более одного пользовательского оператора или оператора с нулификацией. Другими словами, при преобразовании из типа S к типу T никогда не выполняется промежуточное преобразование из типа S к X с последующим преобразованием из X к T.

Точные определения вычисления явных и неявных пользовательских преобразований приведены в последующих разделах. В определениях используются следующие термины:

* Если существует стандартное неявное преобразование (§6.3.1) из типа A к типу B, и при этом типы A и B не являются типами интерфейса, полагается, что тип A включен в тип B, а тип B включает тип A.
* Наиболее охватывающий тип в наборе типов — это один тип, который охватывает все остальные типы в наборе. Если ни один тип не включает все остальные типы, включающий тип верхнего уровня в наборе отсутствует. Другими словами, включающий тип верхнего уровня — это «самый широкий» тип в наборе или тип, в который может быть неявно преобразован любой другой тип набора.
* Включаемым типом нижнего уровня в наборе типов является тип, который включается всеми остальными типами набора. Если ни один тип не включается всеми остальными типами, включаемый тип нижнего уровня в наборе отсутствует. Другими словами, включаемый тип нижнего уровня — это «самый узкий» тип в наборе или тип, который может быть неявно преобразован в любой другой тип набора.

### Определенные пользователем неявные преобразования

Неявное пользовательское преобразование из типа S в тип T выполняется следующим образом.

* Определяются типы S0 и T0. Если типы S и T являются обнуляемыми, типы S0 и T0 являются их базовыми типами; в противном случае S0 и T0 считаются равными S и T соответственно.
* Поиск набора типов, D, на основании которых будут рассматриваться пользовательские операторы преобразования. Этот набор включает в себя тип S0 (если S0 является классом или структурой), базовые классы для S0 (если S0 является классом) и тип T0 (если T0 является классом или структурой).
* Поиск набора применимых пользовательских операторов и операторов преобразования с нулификацией, U. Этот набор состоит из объявленных классами или структурами из D пользовательских операторов и неявных операторов преобразования с нулификацией, которые выполняют преобразование из типа, включающего S, в тип, включаемый типом T. Если набор U является пустым, преобразование считается неопределенным и порождает ошибки времени компиляции.
* Поиск наиболее подходящего исходного типа (SX) для операторов в наборе U:
* Если хотя бы один оператор из набора U выполняет преобразование из S, тип SX равен типу S.
* В противном случае в качестве типа SX выбирается включаемый тип нижнего уровня из объединенного набора исходных типов для операторов набора U. Если найдено несколько включаемых типов нижнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Поиск наиболее подходящего конечного типа (TX) для операторов в наборе U:
* Если хотя бы один оператор из набора U выполняет преобразование в T, тип TX равен типу T.
* В противном случае в качестве типа TX выбирается включающий тип верхнего уровня из объединенного набора конечных типов для операторов набора U. Если найдено несколько включающих типов верхнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Поиск наиболее подходящего оператора преобразования:
* Если набор U содержит ровно один пользовательский оператор преобразования, выполняющий преобразование из типа SX в тип TX, этот оператор будет наиболее подходящим оператором преобразования.
* В противном случае, если набор U содержит только один оператор преобразования с нулификацией из типа SX к TX, он является наиболее подходящим оператором преобразования.
* В противном случае преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Применение преобразования:
* Если тип S не является типом SX, выполняется стандартное неявное преобразование из S в SX.
* Для преобразования из SX к TX вызывается наиболее подходящий оператор преобразования.
* Если тип TX не является типом T, выполняется стандартное неявное преобразование из TX к T.

### Пользовательские явные преобразования

Явное пользовательское преобразование из типа S в тип T выполняется следующим образом.

* Определяются типы S0 и T0. Если типы S и T являются обнуляемыми, типы S0 и T0 являются их базовыми типами; в противном случае S0 и T0 считаются равными S и T соответственно.
* Поиск набора типов, D, на основании которых будут рассматриваться пользовательские операторы преобразования. Этот набор включает в себя тип S0 (если S0 является классом или структурой), базовые классы для S0 (если S0 является классом), тип T0 (если T0 является классом или структурой) и базовые классы для T0 (если T0 является классом).
* Поиск набора применимых пользовательских операторов и операторов преобразования с нулификацией, U. Этот набор состоит из объявленных классами или структурами из D пользовательских операторов и явных или неявных операторов преобразования с нулификацией, которые выполняют преобразование из типа, включающего тип S или включаемого этим типом, в тип, включающий тип T или включаемый этим типом. Если набор U является пустым, преобразование считается неопределенным и порождает ошибки времени компиляции.
* Поиск наиболее подходящего исходного типа (SX) для операторов в наборе U:
* Если хотя бы один оператор из набора U выполняет преобразование из S, тип SX равен типу S.
* В противном случае, если любой из операторов набора U осуществляет преобразование из типа, который включает тип S, то тип SX является включаемым типом нижнего уровня в объединенном наборе исходных типов для этих операторов. Если не найден включаемый тип нижнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* В противном случае в качестве типа SX выбирается включающий тип верхнего уровня из объединенного набора исходных типов для операторов набора U. Если найдено несколько включающих типов верхнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Поиск наиболее подходящего конечного типа (TX) для операторов в наборе U:
* Если хотя бы один оператор из набора U выполняет преобразование в T, тип TX равен типу T.
* В противном случае, если любой из операторов набора U осуществляет преобразование к типу, который включается типом T, то тип TX является включающим типом верхнего уровня в объединенном наборе конечных типов для этих операторов. Если найдено несколько включающих типов верхнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* В противном случае в качестве типа TX выбирается включаемый тип нижнего уровня из объединенного набора конечных типов для операторов набора U. Если не найден включаемый тип нижнего уровня, преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Поиск наиболее подходящего оператора преобразования:
* Если набор U содержит ровно один пользовательский оператор преобразования, выполняющий преобразование из типа SX в тип TX, этот оператор будет наиболее подходящим оператором преобразования.
* В противном случае, если набор U содержит только один оператор преобразования с нулификацией из типа SX к TX, он является наиболее подходящим оператором преобразования.
* В противном случае преобразование считается неоднозначным и порождает ошибку времени компиляции.
* Применение преобразования:
* Если тип S не является типом SX, выполняется стандартное явное преобразование из S в SX.
* Для преобразования из SX к TX вызывается наиболее подходящий пользовательский оператор преобразования.
* Если тип TX не является типом T, выполняется стандартное явное преобразование из TX к T.

## Преобразования анонимных функций

Выражение анонимного метода или лямбда-выражение классифицируется как анонимная функция (§7.15). Такое выражение не имеет типа, однако может быть неявно преобразовано к совместимому типу делегата или дерева выражений. В частности, анонимная функция F совместима с типом делегата D при соблюдении следующих условий.

* Если F содержит сигнатуру анонимной функции, D и F имеют одинаковое число параметров.
* Если F не содержит сигнатуру анонимной функции, тип D может иметь ноль или более параметров любого типа при условии, что ни одному из параметром D не присвоен модификатор out.
* Если F имеет явно типизированный список параметров, каждый параметр D имеет тот же тип и модификаторы, что и соответствующий параметр F.
* Если F имеет неявно типизированный список параметров, D не содержит параметры ref и out.
* Если тело F является выражением и либо D имеет тип возвращаемого значения void, либо F является асинхронной функцией и D имеет тип возвращаемого значения Task, то при условии, что каждому параметру F задан тип соответствующего параметра D, тело F является допустимым выражением (§7), разрешенным в качестве выражения-оператора (§8.6).
* Если тело F является блоком операторов и либо D имеет тип возвращаемого значения void, либо F является асинхронной функцией и D имеет тип возвращаемого значения Task, то при условии, что каждому параметру F задан тип соответствующего параметра D, тело F является допустимым блоком операторов (§8.2), в котором ни один оператор return не задает выражение.
* Если тело F является выражением и *либо* F не является асинхронной функцией и D имеет тип возвращаемого значения T, отличный от void, *либо* F является асинхронной функцией и D имеет тип возвращаемого значения Task<T>, то, когда для каждого параметра F задается тип соответствующего параметра из D, тело F является допустимым выражением (wrt §7), которое можно неявно преобразовать в T.
* Если тело F является блоком операторов и *либо* F не является асинхронной функцией и D имеет тип возвращаемого значения T, отличный от void, *либо* F является асинхронной функцией и D имеет тип возвращаемого значения Task<T>, то, когда для каждого параметра F задается тип соответствующего параметра из D, тело F является допустимым блоком операторов (wrt §8.2) с недостижимой конечной точкой, в которой каждый оператор return определяет выражение, допускающее неявное преобразование в T.

Для краткости в этом разделе используется краткая форма типов задач Task и Task<T> (§10.14).

Лямбда-выражение F совместимо с типом дерева выражений Expression<D> в том случае, если лямбда-выражение F совместимо с типом делегата D. Обратите внимание, что это условие предназначено только для лямбда-выражений, оно не применяется к анонимным методам.

Определенные лямбда-выражения не могут быть преобразованы в тип дерева выражений: даже если преобразование существует, во время компиляции возникает ошибка. Так происходит, если лямбда-выражение:

* содержит тело блока;
* содержит операторы единичного или составного присваивания;
* содержит динамически привязанное выражение;
* является асинхронным.

В приведенных ниже примерах используется универсальный тип делегата Func<A,R>, который представляет функцию, принимающую аргумент типа A и возвращающую значение типа R:

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

В операциях присваивания

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

типы параметров и возвращаемых значений для каждой анонимной функции определяются на основании типа переменной, которой присваивается такая функция.

В результате первой операции присваивания успешно выполняется преобразование анонимной функции к типу делегата Func<int,int>. Поскольку x имеет заданный тип int, x + 1 является допустимым выражением, которое может быть неявно преобразовано к типу int.

Аналогично, в результате второй операции присваивания успешно выполняется преобразование анонимной функции к типу делегата Func<int,double>, поскольку результат выражения x + 1 (тип int) может быть неявно преобразован к типу double.

Однако в результате третьей операции присвоения возникает ошибка времени компиляции: поскольку x имеет заданный тип double, результат выражения x + 1 (типа double) не может быть неявно преобразован в тип int.

В результате четвертой операции присваивания успешно выполняется преобразование анонимной асинхронной функции к типу делегата Func<int, Task<int>>, поскольку результат выражения x + 1 (тип int) может быть неявно преобразован к типу результата int задачи типа Task<int>.

Анонимные функции могут влиять на разрешение перегрузки, а также использоваться при определении типа. Дополнительные сведения см. в разделе §7.5furth.

### Вычисление преобразований анонимных функций к типам делегата

В результате преобразования анонимной функции к типу делегата создается содержащий ссылку на эту функцию экземпляр делегата, а также (возможно пустой) набор внешних записанных переменных, которые активны во время вычисления. При вызове делегата выполняется тело анонимной функции. Код тела функции выполняется с применением набора внешних записанных переменных, на которые ссылается делегат.

Список вызова делегата, созданного на базе анонимной функции, содержит одну запись. Конечные объект и метод делегата не определены. В частности, не определено, является ли конечный объект делегата объектом null, а также значение this включающей функции-члена или любого другого объекта.

Допускаются (но не являются обязательными) преобразования семантически идентичных анонимных функций с одинаковым (возможно пустым) набором экземпляров внешних переменных к одному типу делегата. В результате таких преобразований возвращается один экземпляр делегата. Термин «семантически идентичные» означает, что выполнение анонимных функций с одинаковыми аргументами во всех случаях дает одинаковые результаты. Это правило позволяет оптимизировать код, аналогичный приведенному ниже.

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));  
 ...  
 }  
}

Поскольку делегаты двух анонимных функций имеют одинаковые (возможно пустые) наборы внешних записанных переменных, а сами функции являются семантически идентичными, при компиляции делегаты могут ссылаться на один конечный метод. Соответственно, компилятор может возвращать одинаковые экземпляры делегата по результатам выполнения выражений обеих анонимных функций.

### Вычисление преобразования анонимной функции к типу дерева выражений

В результате преобразования анонимной функции к типу дерева выражений создается дерево выражений (§4.6). Точнее говоря, в результате вычисления преобразования анонимной функции создается объектная структура, представляющая саму функцию. Точная структура дерева выражений, а также процесс его построения указываются при реализации.

### Пример реализации

В данном разделе описывается возможная реализация преобразования анонимных функций в терминах других конструкций C#. Приведенная реализация базируется на принципах, используемых компилятором Microsoft C#, и не является ни обязательной, ни единственно возможной. Здесь приводится лишь краткое описание преобразований к дереву выражений, поскольку полная семантика таких преобразований в данной спецификации не рассматривается.

Далее в разделе рассматривается несколько примеров кода, содержащих анонимные функции с различными характеристиками. Для каждого примера представлен соответствующий код, в котором используются только другие конструкции C#. В этих примерах подразумевается, что идентификатор D представляет следующий тип делегата:

public delegate void D();

Анонимная функция простейшего вида не записывает внешние переменные:

class Test  
{  
 static void F() {  
 D d = () => { Console.WriteLine("test"); };  
 }  
}

Это можно преобразовать в код создания экземпляра делегата, ссылающегося на создаваемый компилятором статический метод, в который помещается код анонимной функции:

class Test  
{  
 static void F() {  
 D d = new D(\_\_Method1);  
 }

static void \_\_Method1() {  
 Console.WriteLine("test");  
 }  
}

В следующем примере анонимная функция ссылается на члены экземпляра this.

class Test  
{  
 int x;

void F() {  
 D d = () => { Console.WriteLine(x); };  
 }  
}

Это можно преобразовать в создаваемый компилятором метод экземпляра, содержащий код анонимной функции:

class Test  
{  
 int x;

void F() {  
 D d = new D(\_\_Method1);  
 }

void \_\_Method1() {  
 Console.WriteLine(x);  
 }  
}

В этом примере анонимная функция записывает локальную переменную:

class Test  
{  
 void F() {  
 int y = 123;  
 D d = () => { Console.WriteLine(y); };  
 }  
}

Время существования локальной переменной необходимо увеличить как минимум до времени существования делегата анонимной функции. Для этого можно поместить (поднять) локальную переменную в поле создаваемого компилятором класса. В этом случае создание экземпляра локальной переменной (§7.15.5.2) будет соответствовать созданию экземпляра класса компилятором. Обращение к локальной переменной будет соответствовать обращению к полю экземпляра создаваемого компилятором класса. Кроме того, анонимная функция становится методом экземпляра создаваемого компилятором класса:

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);  
 }  
 }  
}

И наконец, следующая анонимная функция записывает this и две локальные переменные с различными значениями времени жизни:

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); };  
 }  
 }  
}

Здесь компилятором создается класс для каждого блока оператора, в котором выполняется запись локальных переменных. За счет этого локальные переменные в каждом блоке имеют различное время существования. Экземпляр \_\_Locals2 — сформированный компилятором класс для внутреннего блока операторов — содержит локальную переменную z и поле, которое ссылается на экземпляр \_\_Locals1. Экземпляр \_\_Locals1 — сформированный компилятором класс для внешнего блока операторов — содержит локальную переменную y и поле, которое ссылается на this включающей функции-члена. Используя эти структуры данных, можно получить доступ ко всем внешним переменным через экземпляр класса \_\_Local2. Код анонимной функции, таким образом, может быть реализован как метод экземпляра этого класса.

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);  
 }  
 }  
}

Способ, аналогичный применяемому здесь для записи локальных переменных, может использоваться при преобразовании анонимных функций к деревьям выражений. Ссылки на создаваемые компилятором объекты могут храниться в дереве выражений, а обращение к локальным переменным может быть представлено как обращение к полям этих объектов. Преимущество такого подхода заключается в том, что локальные переменные с нулификацией (поднятые) могут совместно использоваться делегатами и деревьями выражений.

## Преобразования группы методов

Существует неявное преобразование (§6.1) из группы методов (§7.1) к совместимому типу делегата. Для типа делегата D и выражения E, которое классифицируется как группа методов, неявное преобразование из E к D существует в том случае, если E содержит как минимум один метод, который применим в нормальной форме (§7.5.3.1) к списку аргументов, созданному с использованием типов параметров и модификаторов типа D, как описано далее.

Применение во время компиляции преобразования из группы методов E к типу делегата D описывается далее. Обратите внимание, что существование неявного преобразования из E к D не гарантирует, что его выполнение во время компиляции будет завершено без ошибок.

* Отдельный метод M выбирается соответствующим вызову метода (§7.6.5.1) в форме E(A) со следующими модификациями:
* Список аргументов A представляет собой список выражений, которые классифицируются как переменные с типом и модификатором (ref или out), соответствующими параметрам в списке формальных параметров типа D.
* Допустимыми считаются только те методы, которые применимы в нормальной форме (§7.5.3.1), а не те, которые применимы только в расширенной форме.
* Если алгоритм §7.6.5.1 порождает ошибку, возникает ошибка времени компиляции. В противном случае алгоритм возвращает наиболее подходящий метод M, имеющий столько же параметров, что и тип D. В этом случае преобразование считается существующим.
* Выбранный метод M должен быть совместим (§15.2) с типом делегата D. В противном случае порождается ошибка времени компиляции.
* Если выбранный метод M является методом экземпляра, выражения экземпляра, связанное с E, определяет целевой объект делегата.
* Если выбранный метод M является методом расширения, который определяется посредством обращения к члену выражения экземпляра, такое выражение определяет конечный объект делегата.
* Результатом преобразования является значение типа D, то есть вновь созданный делегат, который ссылается на выбранный метод и конечный объект.

Обратите внимание, что в результате этого процесса может быть создан делегат метода расширения. Это происходит в том случае, если алгоритм §7.6.5.1 не находит метод экземпляра, однако успешно выполняет вызов E(A) в качестве вызова метода расширения (§7.6.5.2). Созданный таким образом делегат записывает метод расширения и его первый аргумент.

Далее приведен пример преобразования группы методов:

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

}  
}

При присваивании d1 выполняется неявное преобразование группы методов F к значению типа D1.

Присвоение d2 показывает, как можно создать делегат метода, который имеет типы параметров меньшей глубины наследования (контрвариантные) и тип возвращаемого значения большей глубины наследования (ковариантный).

Присвоение d3 показывает, что преобразование не существует, если метод неприменим.

Присвоение d4 показывает, что метод должен применяться в своем обычном формате.

Присвоение d5 показывает, что типы параметров и возвращаемых значений делегата и метода могут отличаться только для ссылочных типов.

Как и для любых других явных и неявных преобразований, для явного выполнения преобразования группы методов можно использовать оператор приведения. Таким образом, в примере

object obj = new EventHandler(myDialog.OkClick);

может быть записан как

object obj = (EventHandler)myDialog.OkClick;

Группы методов могут влиять на разрешение перегрузки, а также использоваться при определении типа. Дополнительные сведения см. в разделе §7.5.

Вычисление преобразования группы методов во время выполнения осуществляется в следующем порядке:

* Если выбранный во время компиляции метод представляет собой метод экземпляра или метод расширения, обращение к которому происходит как к методу экземпляра, конечный объект делегата определяется с помощью связанного с E выражения экземпляра:
* Выполняется вычисление выражения экземпляра. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Если выражение экземпляра имеет ссылочный тип, значение, возвращенное выражением экземпляра, становится конечным объектом. Если выбранный метод является методом экземпляра и целевой объект равен null, возникает исключение System.NullReferenceException и последующие действия не выполняются.
* Если выражение экземпляра имеет тип значений, выполняется операция упаковки (§4.3.1) для преобразования значения к объекту. Этот объект становится конечным объектом.
* В противном случае выбранный метод является частью вызова статического метода и целевой объект делегата равен null.
* Создается новый экземпляр типа делегата D. Если для создания нового экземпляра недостаточно памяти, то возникает исключение System.OutOfMemoryException и дальнейшие этапы не выполняются
* Инициализируется новый экземпляр делегата, ссылающийся на метод, который был определен во время компиляции, а также на вычисленный выше конечный объект.

# Выражения

Выражение — это последовательность операторов и операндов. В этой главе описывается синтаксис, порядок вычисления операндов и операторов, а также значение выражений.

## Классы выражений

Выражение может иметь один из следующих классов:

* Значение. У каждого значения есть связанный с ним тип.
* Переменная. У каждой переменной есть связанный с ней тип, а именно объявленный тип переменной.
* Пространство имен. Выражение такого класса может находиться только в левой части доступа к члену (§7.6.4). В любом другом контексте выражение класса пространства имен вызывает ошибку времени компиляции.
* Тип. Выражение такого класса может находиться только в левой части доступа к члену (§7.6.4) или быть операндом оператора as (§7.10.11), is (§7.10.10) или typeof (§7.6.11). В любом другом контексте выражение класса типа вызывает ошибку времени компиляции.
* Группа методов, которая представляет собой набор перегруженных методов, получающихся в результате поиска члена (§7.4). У группы методов может быть связанное выражение экземпляра и связанный список аргументов типа. При вызове метода экземпляра результат вычисления выражения экземпляра становится экземпляром, который представляется оператором this (§7.6.7). Группа методов может быть частью выражения вызова (§7.6.5) или выражения создания делегата (§7.6.10.5), левой стороной оператора is, а также может неявно преобразовываться в совместимый тип делегата (§6.6). В любом другом контексте выражение с классом группы методов вызывает ошибку времени компиляции.
* Литерал null. Выражение такого класса может неявно преобразовываться в ссылочный тип или обнуляемый тип.
* Анонимная функция. Выражение такого класса может неявно преобразовываться в совместимый тип делегата или тип дерева выражения.
* Доступ к свойству. У каждого доступа к свойству есть связанный тип, а именно тип свойства. Кроме того, у доступа к свойству может быть связанное выражение экземпляра. При вызове метода доступа (блок get или set) в доступе к свойству экземпляра результат вычисления выражения экземпляра становится экземпляром, который представляется оператором this (§7.6.7).
* Доступ к событию. У каждого доступа к событию есть связанный тип, а именно тип события. Кроме того, у доступа к событию может быть связанное выражение экземпляра. Доступ к событию может быть левым операндом операторов += и -= (§7.17.3). В любом другом контексте выражение с классом доступа к событию вызывает ошибку времени компиляции.
* Доступ к индексатору. У каждого доступа к индексатору есть связанный тип, а именно тип индексатора. Кроме того, у доступа к индексатору есть связанное выражение экземпляра и связанный список аргументов. При вызове метода доступа (блок get или set) в доступе к индексатору результат вычисления выражения экземпляра становится экземпляром, который представляется оператором this (§7.6.7), а результат вычисления списка аргументов становится списком параметров вызова.
* Класс отсутствует. Это происходит, когда выражение является вызовом метода с типом возвращаемого значения void. Выражение без класса допустимо только в контексте выражения оператора (§8.6).

Конечным результатом выражения не может быть пространство имен, тип, группа методов или доступ к событию. Наоборот, как указано выше, данные категории выражений представляют собой промежуточные конструкции, которые допустимы только в определенных контекстах.

Доступ к свойству или доступ к индексатору всегда изменяет свой класс на значение при вызове метода доступа get или set. Конкретный метод доступа определяется контекстом доступа к свойству или индексатору: если доступ является назначением присваивания, то для назначения нового значения вызывается метод доступа set (§7.17.1). В противном случае для получения текущего значения вызывается метод доступа get (§7.1.1).

### Значения выражений

В большинстве конструкций, в которые входят выражения, в конечном счете требуется, чтобы выражение представляло собой значение. В таких случаях, если фактическое выражение обозначает пространство имен, тип, группу методов или не имеет значения, возникает ошибка времени компиляции. Однако если выражение обозначает доступ к свойству или индексатору или переменную, то происходит неявная подстановка значения свойства, индексатора или переменной.

* Значение переменной — это просто значение, которое хранится в данный момент в расположении, указанном переменной. Для получения значения переменной оно должно быть явно присвоено (§5.3), в противном случае возникает ошибка времени компиляции.
* Значение выражения доступа к свойству получается путем вызова метода доступа get для свойства. Если у свойства нет метода доступа get, возникает ошибка времени компиляции. В противном случае выполняется вызов функции-члена (§7.5.4), и значением выражения доступа к свойству становится результат этого вызова.
* Значение выражения доступа к индексатору получается путем вызова метода доступа get для индексатора. Если у индексатора нет метода доступа get, возникает ошибка времени компиляции. В противном случае выполняется вызов функции-члена (§7.5.4) со списком аргументов, связанных с выражением доступа к индексатору, и значением выражения доступа к индексатору становится результат этого вызова.

## Статическая и динамическая привязка

Процесс определения значения операции на основе типа или значения составляющих выражений (аргументы, операнды, выражения-получатели) часто называют привязкой. Например, значение вывода метода определяется на основе типа выражения-получателя или аргументов. Значение оператора определяется на основе типа его операндов.

В C# значения операции обычно определяется во время компиляции на основе типа времени компиляции его составляющих выражений. Также, если выражение содержит ошибку, она определяется и объявляется компилятором. Подобный метод называется статической привязкой.

Однако если выражение является динамическим (т. е. имеет тип dynamic), это указывает на то, что любая участвующая привязка должна быть основана на типе времени выполнения (т. е. фактическом типе объекта, означаемого временем выполнения), а не на типе, который она имела во время компиляции. Привязка такой операции откладывается до времени выполнения операции при работающей программе. Это называется динамической привязкой.

При динамической привязке операции компилятором либо выполняется краткая проверка, либо не выполняется никакой проверки. Вместо этого, при ошибках привязки во время выполнения они объявляются как исключения во время выполнения.

В C# имеются следующие операции привязки:

* Доступ к члену: e.M
* Вызов метода: e.M(e1,…,en)
* Вызов делегата: e(e1,…,en)
* Доступ к элементу: e[e1,…,en]
* Создание объектов: new C(e1,…,en)
* Перегруженные унарные операторы: +, -, !, ~, ++, --, true, false
* перегруженные бинарные операторы: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==,!=, >, <, >=, <=
* Операторы присваивания: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
* явные и неявные преобразования.

Если динамические выражения не включены, C# устанавливает статическую привязку, т. е. типы времени компиляции составляющих выражений используются в процессе выбора. Однако если одно из составляющих выражений в операциях, перечисленных выше, является динамическим выражением, операция привязывается динамически.

### Время привязки

Статическая привязка выполняется во время компиляции, а динамическая – в процессе выполнения. В следующих разделах термин время привязки относится как ко времени компиляции, так и ко времени выполнения в зависимости от того, когда выполняется привязка.

Следующий пример иллюстрирует уведомления статической и динамической привязок, а также время привязки.

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)

Два первых вызова являются статическими: перегрузка Console.WriteLine выбрана на основе типа времени компиляции аргумента. Таким образом, время привязки здесь — время компиляции.

Третий вызов является динамическим: перегрузка Console.WriteLine выбрана на основе типа времени выполнения аргумента. Это происходит по причине того, что аргумент является динамическим выражением — его тип времени компиляции — dynamic. Таким образом, время привязки третьего вызова — время выполнения.

### Динамическая привязка

Назначение динамической привязки – позволить программам C# взаимодействовать с динамическими объектами, т. е. объектами, которые не подчиняются обычным правилам системы типов C#. Динамическими могут быть объекты других языков программирования с другими системами типов или объекты, которые запрограммированы на реализацию собственной семантики привязки для различных операций.

Механизм, по которому динамический объект реализует собственную семантику, определяется во время реализации. Предоставляемый интерфейс – также определяемый во время реализации – реализуется динамическими объектами для указания во время выполнения C#, что они имею собственную семантику. Таким образом, когда бы не выполнялась динамическая привязка операций и динамических объектов, преимущество имеет их собственная семантика привязки, а не та, что указана в данном документе C#.

Хотя назначение динамической привязки – взаимодействие с динамическими объектами, C# позволяет выполнять динамическую привязку для всех объектов, вне зависимости от того, являются они динамическими или нет. Это делает возможным более гладкую реализацию динамических объектов, так как результаты операций над ними могут сами по себе не являться динамическими объектами, а быть результатами типа, неизвестного программисту во время компиляции. Также динамическая привязка полезна для устранения кода на основе отражений, приводящего к возникновению ошибок, даже когда среди вовлеченных объектов нет динамических.

В следующих разделах представлены описания для каждой конструкции в языке: когда применяется динамическая привязка, какая выполняется проверка во время компиляции (и выполняется ли) и какая используется классификация результатов времени компиляции и выражений.

### Типы составных выражений

Если операция имеет статическую привязку, тип составного выражения (например, получатель, аргумент and, индекс или операнд) всегда считается типом времени компиляции данного выражения.

Если операция имеет динамическую привязку, тип составного выражения определяется по-разному в зависимости от типа времени компиляции составного выражения:

* составное выражение типа времени компиляции dynamic должно иметь тип фактического значения, которое выражение получает во время выполнения;
* составное выражение, чей тип времени компиляции является параметром типа, должно иметь тип, к которому параметр типа привязан во время выполнения;
* в противном случае составное выражение должно иметь свой тип времени компиляции.

## Операторы

Выражения состоят из операндов и операторов. Операторы в выражении указывают, какие операции производятся с операндами. К операторам относятся, например, +, -, \*, / и new. К операндам относятся, например, литералы, поля, локальные переменные и выражения.

Существует три типа операторов.

* Унарные операторы. У унарного оператора есть только один операнд и оператор может записываться в префиксной форме (например, –x) или постфиксной форме (например, x++).
* Бинарные операторы. У бинарных операторов два операнда и они записываются в виде инфикса (например, x + y).
* Тернарный оператор. Существует только один тернарный оператор, ?:. В нем три операнда и используется инфиксная запись (c? x: y).

Порядок вычисления операторов в выражении определяется приоритетом и ассоциативностью операторов (§7.3.1).

Операнды в выражении вычисляются слева направо. Например, в выражении F(i) + G(i++) \* H(i) вызывается метод F со старым значением i, затем вызывается метод G со старым значением i и, наконец, вызывается метод H с новым значением i. Это никак не связано с приоритетом операторов.

Некоторые операторы допускают перегрузку. Перегрузка операторов позволяет использовать пользовательскую реализацию операторов в операциях, в которых один или оба операнда имеют пользовательский тип класса или структуры (§7.3.2).

### Приоритет и ассоциативность операторов

Когда выражение содержит несколько операторов, порядок вычисления отдельных операторов задается приоритетом операторов. Например, выражение x + y \* z вычисляется как x + (y \* z), поскольку оператор \* имеет более высокий приоритет по сравнению с бинарным оператором +. Приоритет оператора задается в определении связанной с ним грамматической структуры. Например, аддитивное выражение состоит из последовательности мультипликативных выражений, разделенных операторами + или -, таким образом, операторы + или - имеют более низкий приоритет, чем операторы \*, / и %.

В следующей таблице приводятся все операторы в соответствии с их приоритетом от самого высокого до самого низкого.

|  |  |  |
| --- | --- | --- |
| **Раздел** | **Категория** | **Операторы** |
| 7.6 | Основной | x.y f(x) a[x] x++ x-- new  typeof default checked unchecked delegate |
| 7.7 | Унарный | + - ! ~ ++x --x (T)x |
| 7.8 | Мультипликативный | \* / % |
| 7.8 | Аддитивный | + - |
| 7.9 | Сдвиг | << >> |
| 7.10 | Отношение и проверка типа | < > <= >= is as |
| 7.10 | Равенство | == != |
| 7.11 | Логическое И | & |
| 7.11 | Исключающее ИЛИ | ^ |
| 7.11 | Логическое ИЛИ | | |
| 7.12 | Условное И | && |
| 7.12 | Условное ИЛИ | || |
| 7.13 | Объединение с нулем | ?? |
| 7.14 | Условный | ?: |
| 7.17, 7.15 | Присваивание и лямбда-выражение | = \*= /= %= += -= <<= >>= &= ^= |=  => |

Когда операнд находится между двух операторов с одним приоритетом, порядок выполнения операций определяется ассоциативностью операторов.

* За исключением операторов присваивания и оператора слияния с NULL, все бинарные операторы обладают левой ассоциативностью. Это означает, что все операции выполняются слева направо. Например, выражение x + y + z вычисляется как (x + y) + z.
* Операторы присваивания, оператор слияния с NULL и условный оператор (?:) обладают правой ассоциативностью. Это означает, что все операции выполняются справа налево. Например, выражение x = y = z вычисляется как x = (y = z).

Приоритетом и ассоциативностью можно управлять с помощью скобок. Например, в выражении x + y \* z сначала y умножается на z, а затем результат складывается с x, но в выражении (x + y) \* z сначала складываются x и y, а затем результат умножается на z.

### Перегрузка операторов

У всех унарных и бинарных операторов есть стандартная реализация, доступная автоматически в любом выражении. В дополнение к стандартным реализациям объявление operator в классах и структурах позволяет использовать пользовательские реализации (§10.10). Пользовательские реализации операторов всегда имеют приоритет над предопределенными реализациями операторов: только в случае, если не существует пользовательских реализаций операторов, будут рассматриваться предопределенные реализации операторов, как описано в §7.3.3 and §7.3.4.

К унарным операторам, допускающим перегрузку, относятся:

+ - ! ~ ++ -- true false

Несмотря на то что true и false явно в выражениях не используются (и поэтому не включены в таблицу с приоритетами в разделе §7.3.1), они считаются операторами, потому что вызываются в некоторых контекстах выражений: логические выражения (§7.20), выражения, включающие условия (§7.14), и логические условные операторы (§7.12).

К бинарным операторам, допускающим перегрузку, относятся:

+ - \* / % & | ^ << >> == != > < >= <=

Перегрузка возможна только для операторов, указанных выше. В частности, невозможна перегрузка операторов доступа к членам, вызова методов или операторов =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as и is.

При перегрузке бинарного оператора связанный оператор присваивания (если он есть) также неявно перегружается. Например, при перегрузке оператора \* также выполняется перегрузка оператора \*=. Это описано далее в §7.17.2. Обратите внимание, что сам оператор присваивания (=) нельзя перегрузить. При присваивании всегда происходит простое побитовое копирование значения в переменную.

Перегрузка операций приведения типов, например (T)x, осуществляется путем предоставления пользовательских преобразований (§6.4).

Доступ к элементу, например a[x], не считается оператором, допускающим перегрузку. Вместо этого поддерживается пользовательская индексация в индексаторах (§10.9).

В выражениях ссылка на операторы осуществляется с помощью записи в виде оператора, а в объявлениях ссылка на операторы осуществляется с помощью записи в виде функции. В следующей таблице показано отношение между записью в виде оператора и функции для унарных и бинарных операторов. В первой строке op означает любой унарный оператор в виде префикса, допускающий перегрузку. Во второй строке op означает унарный оператор ++ или -- в виде постфикса. В третьей строке op означает любой бинарный оператор, допускающий перегрузку.

|  |  |
| --- | --- |
| **Запись в виде оператора** | **Запись в виде функции** |
| op x | operator op(x) |
| x op | operator op(x) |
| x op y | operator op(x, y) |

В пользовательских объявлениях операторов по крайней мере один из параметров всегда должен иметь тип класса или структуры, в котором содержится объявление оператора. Таким образом, у пользовательского оператора не может быть такой же сигнатуры, что и у стандартного оператора.

В объявлениях пользовательских операторов нельзя изменять синтаксис, приоритет и ассоциативность оператора. Например, оператор / всегда будет бинарным оператором, иметь приоритет, указанный в разделе §7.3.1 и всегда будет обладать левой ассоциативностью.

Хотя пользовательский оператор может выполнять любые вычисления, настоятельно не рекомендуется создавать реализации, результат которых не является интуитивно ожидаемым. Например, в реализации operator == должно производиться сравнение двух операндов на равенство и возвращаться соответствующий результат типа bool.

В описании отдельных операторов в разделах §7.6–§7.12 указаны стандартные реализации и дополнительные правила, применимые к каждому оператору. В этих описаниях используются термины разрешение перегрузки унарных операторов, разрешение перегрузки бинарных операторов и числовое расширение, определения которых находятся в дальнейших разделах.

### Разрешение перегрузки унарных операторов

Операция вида op x или x op, где op — это унарный оператор, допускающий перегрузку, а x — это выражение типа X, выполняется следующим образом.

* С помощью правил из раздела §7.3.5 определяется набор пользовательских операторов-кандидатов, предоставленных в типе X для операции operator op(x).
* Если набор пользовательских операторов-кандидатов не пустой, то он становится набором операторов-кандидатов для операции. В противном случае набором операторов-кандидатов для операции становятся стандартные реализации унарного оператора operator op, включая их варианты с нулификацией. Стандартные реализации данного оператора указываются в описании оператора (§7.6 и §7.7).
* Чтобы выбрать оператор, более всего подходящий для списка аргументов (x), к набору операторов-кандидатов применяются правила разрешения перегрузки из раздела §7.5.3, и этот оператор становится результатом процесса разрешения перегрузки. Если при разрешении перегрузки не удалось выбрать один подходящий оператор, то возникает ошибка времени привязки.

### Разрешение перегрузки бинарных операторов

Операция вида x op y, где op — это бинарный оператор, допускающий перегрузку, x — это выражение типа X, а y — это выражение типа Y, выполняется следующим образом.

* Определяется набор пользовательских операторов-кандидатов, предоставленных типами X и Y для операции operator op(x, y). Этот набор представляет собой объединение операторов-кандидатов, предоставленных в типе X, и операторов-кандидатов, предоставленных в типе Y. Каждый из этих составляющих наборов определяется в соответствии с правилами из раздела §7.3.5. Если X и Y представляют один тип или X и Y являются производными от общего базового типа, тогда в объединенном наборе общие операторы-кандидаты появляются только один раз.
* Если набор пользовательских операторов-кандидатов не пустой, то он становится набором операторов-кандидатов для операции. В противном случае набором операторов-кандидатов для операции становятся стандартные реализации бинарного оператора operator op, включая их варианты с нулификацией. Стандартные реализации данного оператора указываются в описании оператора (§7.8–§7.12). Для предопределенных операторов перечислений и делегатов рассматриваются только операторы, определяемые типом перечисления или делегата, который является типом времени привязки одного из операндов.
* Чтобы выбрать оператор, более всего подходящий для списка аргументов (x, y), к набору операторов-кандидатов применяются правила разрешения перегрузки из раздела §7.5.3, и этот оператор становится результатом процесса разрешения перегрузки. Если при разрешении перегрузки не удалось выбрать один подходящий оператор, то возникает ошибка времени привязки.

### Пользовательские операторы-кандидаты

Если взять тип T и операцию operator op(A), где op — это оператор, допускающий перегрузку, а A — это список аргументов, то набор пользовательских операторов-кандидатов, предоставляемых типом T для оператора operator op(A), определяется следующим образом.

* Определяется тип T0. Если тип T допускает значения NULL, T0 является его базовым типом, в противном случае T0 совпадает с T.
* Для всех объявлений operator op в T0 и всех вариантов таких операторов с нулификацией, если в соответствии со списком аргументов A применим хотя бы один оператор (§7.5.3.1), то набор операторов-кандидатов состоит из всех таких применимых операторов в T0.
* В противном случае, если T0 является типом object, набор операторов-кандидатов является пустым.
* Иначе набор операторов-кандидатов, предоставленных в T0, является набором операторов-кандидатов, предоставленных в непосредственном базовом классе для T0 или в фактическом базовом классе для T0, если T0 является параметром типа.

### Числовое расширение

Числовое расширение — это автоматическое проведение определенных неявных преобразований операндов в стандартных унарных и бинарных числовых операторах. Числовое расширение — это не отдельный механизм, а скорее эффект от проведения разрешения перегрузки для стандартных операторов. Числовое расширение специально не влияет на вычисление пользовательских операторов, хотя пользовательские операторы можно реализовать для получения похожих эффектов.

В качестве примера числового расширения можно рассмотреть стандартные реализации бинарного оператора \*:

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);

При применении правил разрешения перегрузки (§7.5.3) к этому набору операторов результатом является первый оператор, для которого существуют неявные преобразования операндов. Например, для операции b \* s, где b имеет тип byte, а s — тип short, при разрешении перегрузки в качестве наилучшего оператора будет выбран operator \*(int, int). Таким образом, b и s преобразуются в тип int, и результат имеет тип int. Аналогичным образом, для операции i \* d, где i имеет тип int, а d — тип double, при разрешении перегрузки в качестве наилучшего оператора будет выбран operator \*(double, double).

#### Числовое расширение унарных операторов

Числовое расширение унарных операторов выполняется для операндов стандартных унарных операторов +, – и ~. Числовое расширение унарных операторов заключается в простом преобразовании операндов типа sbyte, byte, short, ushort или char в тип int. Кроме того, для унарного оператора – при числовом расширении унарного оператора происходит преобразование операндов с типом uint в тип long.

#### Числовое расширение бинарных операторов

Числовое расширение бинарных операторов выполняется для операндов стандартных бинарных операторов +, –, \*, /, %, &, |, ^, ==, !=, >, <, >= и <=. При числовом расширении бинарных операторов оба операнда преобразуются в общий тип. Для операторов, отличных от операторов отношения, этот тип становится типом результата операции. Числовое расширение бинарных операторов состоит в применении следующих правил в указанном здесь порядке.

* Если один из операндов имеет тип decimal, то другой операнд преобразуется в тип decimal, либо если другой операнд имеет тип float или double, то возникает ошибка времени привязки.
* Иначе, если один из операндов имеет тип double, то другой операнд преобразуется в тип double.
* Иначе, если один из операндов имеет тип float, то другой операнд преобразуется в тип float.
* В противном случае, если один из операндов имеет тип ulong, то другой операнд преобразуется в тип ulong, либо, если другой операнд имеет тип sbyte, short, int или long, возникает ошибка времени привязки.
* Иначе, если один из операндов имеет тип long, то другой операнд преобразуется в тип long.
* В противном случае, если один из операндов имеет тип uint, а другой операнд — тип sbyte, short или int, оба операнда преобразуются в тип long.
* Иначе, если один из операндов имеет тип uint, то другой операнд преобразуется в тип uint.
* Иначе оба операнда преобразуются в тип int.

Обратите внимание, что первое правило запрещает любые операции, в которых смешиваются тип decimal и типы double и float. Это правило вытекает из того факта, что неявное преобразование между типом decimal и типом double или float не существует.

Также обратите внимание, что операнд не может иметь тип ulong, когда другой операнд имеет целый тип со знаком. Это так, поскольку не существует целого типа, который может представлять полный спектр значений ulong вместе с целыми типами со знаком.

В обоих указанных случаях для явного преобразования одного операнда в тип, совместимый с другим операндом, можно использовать выражение приведения типа.

В этом примере

decimal AddPercent(decimal x, double percent) {  
 return x \* (1.0 + percent / 100.0);  
}

возникает ошибка времени привязки, поскольку значения типа decimal нельзя умножать на значение типа double. Эта ошибка устраняется с помощью явного преобразования второго операнда в тип decimal, ср.:

decimal AddPercent(decimal x, double percent) {  
 return x \* (decimal)(1.0 + percent / 100.0);  
}

### Операторы с нулификацией

Операторы с нулификацией позволяют использовать стандартные и пользовательские операторы, применимые для необнуляемых типов значений, со значениями обнуляемых типов. Операторы с нулификацией создаются на основе стандартных и пользовательских операторов, которые соответствуют определенным требованиям, описанным ниже.

* Для унарных операторов

+ ++ - -- ! ~

вариант оператора с нулификацией существует, если типы операнда и результата оба являются необнуляемыми типами значений. Вариант оператора с нулификацией создается с помощью добавления одного модификатора ? к типу операнда и результата. Значением оператора с нулификацией является значение null, если операнд равен null. В противном случае в операторе с нулификацией с операнда снимается упаковка, применяется базовый оператор и затем создается упаковка для результата.

* Для бинарных операторов.

+ - \* / % & | ^ << >>

вариант оператора с нулификацией существует, если типы операнда и результата являются необнуляемыми типами значений. Вариант оператора с нулификацией создается с помощью добавления одного модификатора ? к типу каждого операнда и результата. Значением оператора с нулификацией является значение NULL, если один или оба операнда равны NULL (исключение составляют операторы & и | типа bool?, см. раздел §7.11.3). В противном случае в операторе с нулификацией с операндов снимается упаковка, применяется базовый оператор и затем создается упаковка для результата.

* Для операторов равенства

== !=

вариант оператора с нулификацией существует, если оба операнда имеют необнуляемый тип значений, а результат имеет тип bool. Вариант оператора с нулификацией создается с помощью добавления одного модификатора ? к каждому типу операнда. В операторе с нулификацией равными считаются два значения null, а неравными — значение null и любое другое ненулевое значение. Если оба операнда не равны NULL, то в операторе с нулификацией с операндов снимается упаковка и для получения результата с типом bool применяется базовый оператор.

* Для операторов отношения

< > <= >=

вариант оператора с нулификацией существует, если оба операнда имеют необнуляемый тип значений, а результат имеет тип bool. Вариант оператора с нулификацией создается с помощью добавления одного модификатора ? к каждому типу операнда. Значением оператора с нулификацией является false, если один или оба операнда имеют значение NULL. В противном случае в операторе с нулификацией с операндов снимается упаковка и для получения результата с типом bool применяется базовый оператор.

## Поиск членов

Поиск членов — это процесс, при котором определяется значение имени в контексте данного типа. Поиск членов может выполняться в составе процесса вычисления простого имени (§7.6.2) или доступа к члену (§7.6.4) в выражении. Если простое имя или доступ к члену указывается в виде простого выражения для выражения вызова (§7.6.5.1), то говорят, что этот член вызываемый.

Если член является методом или событием или если он является константой, полем или свойством либо типа делегата (§15), либо типа dynamic (§4.7), то говорят, что этот метод допускает вызов.

При поиске членов учитывается не только имя члена, но также и число параметров типа, имеющихся у члена, а также, доступен ли член. Для поиска членов в универсальных методах и во вложенных универсальных методах в соответствующих объявлениях указывается определенное число параметров типа, а у всех остальных членов параметров типа нет.

Поиск члена с именем N и параметрами типа K в типе T происходит следующим образом.

* Сначала определяется набор доступных членов с именем N.
* Если T — это параметр типа, то такой набор представляет собой объединение наборов доступных членов с именем N из каждого типа, указанного в качестве первичного или вторичного ограничения (§10.1.5) для T, вместе с набором доступных членов с именем N в object.
* В противном случае набор состоит из всех доступных членов (§3.5) с именем N в типе T, включая унаследованные члены и доступные члены с именем N в типе object. Если T — это сформированный тип, то набор членов получается заменой аргументов типа, как описано разделе в §10.3.2. Из набора исключаются члены с модификатором override.
* Затем, если K равно нулю, удаляются все вложенные типы, в объявления которых входят параметры типа. Если K не равно нулю, удаляются все члены с отличающимся числом параметров типа. Обратите внимание, что когда K равно нулю, методы с параметрами типа не удаляются, поскольку в процессе определения типа (§7.5.2) могут выводиться аргументы типа.
* Затем, если метод является вызываемым, из набора удаляются все не допускающие вызов члены.
* Затем из набора удаляются члены, скрытые другими членами. Для каждого члена S.M в наборе, где S — это тип, в котором определен член M, применяются следующие правила.
* Если M является константой, полем, свойством, событием или элементом перечисления, из набора удаляются все члены, объявленные в базовом типе S.
* Если M является объявлением типа, из набора удаляются все объявленные в базовом типе S члены, не являющиеся типами, а также все объявления типов с таким же числом параметров типа, как и в M, объявленные в базовом типе S.
* Если M является методом, то из набора удаляются все объявленные в базовом типе S члены, не являющиеся методами.
* Затем из набора удаляются члены интерфейса, скрытые членами класса. Этот шаг выполняется, только если T является параметром типа и у T есть как действительный базовый класс, отличный от класса object, так и действительный непустой набор интерфейсов (§10.1.5). Для каждого члена S.M в наборе, где S — это тип, в котором определен член M, применяются следующие правила, если S является объявлением класса, отличного от object.
* Если M является константой, полем, свойством, событием, элементом перечисления или объявлением типа, из набора удаляются все члены, объявленные в объявлении интерфейса.
* Если M является методом, из набора удаляются все объявленные в объявлении интерфейса члены, не являющиеся методами, а также все объявленные в объявлении интерфейса методы с такой же подписью, как и у M.
* Наконец, после удаления скрытых членов определяется результат поиска.
* Если набор состоит из одного члена, который не является методом, то этот член является результатом поиска.
* Иначе, если набор содержит только методы, то эта группа методов является результатом поиска.
* Иначе поиск привел к неоднозначным результатам, что вызывает ошибку времени привязки.

При поиске членов в типах, отличных от параметров типа и интерфейсов, а также поиске членов в интерфейсах со строго единственным наследованием (каждый интерфейс в цепочке наследования имеет единственный непосредственный базовый интерфейс или не имеет такого) в результате применения правил поиска происходит простое скрытие базовых членов производными членами с такими же именами. При поиске в случае строго единственного наследования неоднозначности никогда не возникает. Случаи неоднозначности, которые могут возникать при поиске членов в случае с множественным наследованием, описываются в разделе §13.2.5.

### Базовые типы

Для поиска членов считается, что у типа T есть следующие базовые типы:

* Если T имеет тип object, у T нет базового типа.
* Если T имеет тип перечисления, то базовыми типами T являются типы классов System.Enum, System.ValueType и object.
* Если T имеет тип структуры, то базовыми типами T являются типы классов System.ValueType и object.
* Если T имеет тип класса, то базовыми классами T являются базовые классы T, включая тип класса object.
* Если T имеет тип интерфейса, то базовыми классами T являются базовые интерфейсы T и тип класса object.
* Если T имеет тип массива, то базовыми типами T являются типы классов System.Array и object.
* Если T имеет тип делегата, то базовыми типами T являются типы классов System.Delegate и object.

## Функции-члены

Функции-члены — это члены, содержащие исполняемые операторы. Функции-члены всегда являются членами типов и не могут быть членами пространств имен. В C# определены следующие категории функций-членов:

* Методы
* Свойства
* События
* Индексаторы
* Пользовательские операторы
* Конструкторы экземпляров
* Статические конструкторы
* Деструкторы

За исключением деструкторов и статических конструкторов (которые нельзя вызывать явно) операторы в функциях-членах выполняются при вызовах функции-члена. Фактический синтаксис вызова функции-члена зависит от конкретной категории функции-члена.

Список аргументов (§7.5.1) для функции-члена содержит фактические значения или ссылки на переменные для параметров функции-члена.

Вызовы универсальных методов могут использовать выводы типа для определения набора типов аргументов для выполнения метода. Этот процесс описывается в разделе §7.5.2.

При вызове методов, индексаторов, операторов и конструкторов экземпляров выполняется разрешение перегрузки с целью определить, какой набор кандидатов функций-членов будет вызываться. Этот процесс описывается в разделе §7.5.3.

После определения конкретной функции-члена во время привязки (возможно, в процессе разрешения перегрузки), фактический процесс вызова функции-члена во время выполнения описывается в разделе §7.5.4.

В следующей таблице приводится сводка по операциям, которые выполняются в конструкциях с шестью категориями функций-членов, допускающих явный вызов. В таблице e, x, y и value обозначают выражения с типом переменной или значения, T означает выражение типа, F — это простое имя метода, а P — это простое имя свойства.

| **Конструкция** | **Пример** | **Описание** |
| --- | --- | --- |
| Вызов метода | F(x, y) | Для выбора наиболее подходящего метода F в содержащем классе или структуре используется разрешение перегрузки. Метод вызывается со списком аргументов (x, y). Если у метода нет модификатора static, выражением экземпляра является this. |
| T.F(x, y) | Для выбора наиболее подходящего метода F в классе или структуре T используется разрешение перегрузки. Если метод не относится к типу static, возникает ошибка времени привязки. Метод вызывается со списком аргументов (x, y). |
| e.F(x, y) | Для выбора наиболее подходящего метода F в классе, структуре или интерфейсе, передаваемом в типе e используется разрешение перегрузки. Если метод относится к типу static, возникает ошибка времени привязки. Метод вызывается с выражением экземпляра e и со списком аргументов (x, y). |
| Доступ к свойству | P | В содержащем классе или структуре вызывается метод доступа get свойства P. Если свойство P доступно только для записи, то возникает ошибка времени компиляции. Если у P нет модификатора static, выражением экземпляра является this. |
| P = value | В содержащем классе или структуре вызывается метод доступа set свойства P со списком аргументов (value). Если свойство P доступно только для чтения, то возникает ошибка времени компиляции. Если у P нет модификатора static, выражением экземпляра является this. |
| T.P | В классе или структуре T вызывается метод доступа get свойства P. Если у свойства P нет модификатора static или свойство P доступно только для записи, возникает ошибка времени компиляции. |
| T.P = value | В классе или структуре T вызывается метод доступа set свойства P со списком аргументов (value). Если у свойства P нет модификатора static или свойство P доступно только для чтения, возникает ошибка времени компиляции. |
| e.P | В классе, структуре или интерфейсе, передаваемом по типу e, вызывается метод доступа get свойства P с выражением экземпляра e. Если свойству P присвоен модификатор static или свойство P доступно только для записи, возникает ошибка времени привязки. |
| e.P = value | В классе, структуре или интерфейсе, передаваемом по типу e, вызывается метод доступа set свойства P с выражением экземпляра e и списком аргументов (value). Если свойству P присвоен модификатор static или свойство P доступно только для чтения, возникает ошибка времени привязки. |
| Доступ к событию | E += value | В содержащем классе или структуре вызывается метод доступа add события E. Если событие E не относится к типу static, то выражением экземпляра является this. |
| E -= value | В содержащем классе или структуре вызывается метод доступа remove события E. Если событие E не относится к типу static, то выражением экземпляра является this. |
| T.E += value | В классе или структуре T вызывается метод доступа add события E. Ошибка времени привязки возникает, если E не является статическим. |
| T.E -= value | В классе или структуре T вызывается метод доступа remove события E. Ошибка времени привязки возникает, если E не является статическим. |
| e.E += value | В классе, структуре или интерфейсе, передаваемом по типу e, вызывается метод доступа add события E с выражением экземпляра e. Если событие E не относится к типу static, возникает ошибка времени привязки. |
| e.E -= value | В классе, структуре или интерфейсе, передаваемом по типу e, вызывается метод доступа remove события E с выражением экземпляра e. Если событие E не относится к типу static, возникает ошибка времени привязки. |
| Доступ к индексатору | e[x, y] | Разрешение перегрузки применяется для выбора лучшего индексатора в классе, структуре или интерфейсе, предоставленном типом e. Метод доступа get индексатора вызывается выражением экземпляра e и списком аргументов (x, y). Если индексатор доступен только на запись, то возникает ошибка времени привязки. |
| e[x, y] = value | Для выбора наиболее подходящего индексатора в классе, структуре или интерфейсе, передаваемом в типе e, используется разрешение перегрузки. Вызывается метод доступа set индексатора с выражением экземпляра e и со списком аргументов (x, y, value). Если индексатор доступен только на чтение, то возникает ошибка времени привязки. |
| Вызов оператора | -x | Для выбора наиболее подходящего унарного оператора в классе или структуре, передаваемой в типе x, используется разрешение перегрузки. Выбранный оператор вызывается со списком аргументов (x). |
| x + y | Для выбора наиболее подходящего бинарного оператора в классах или структурах, передаваемых в типах x и y, используется разрешение перегрузки. Выбранный оператор вызывается со списком аргументов (x, y). |
| Вызов конструктора экземпляра | new T(x, y) | Для выбора наиболее подходящего конструктора экземпляра в классе или структуре T используется разрешение перегрузки. Конструктор экземпляра вызывается со списком аргументов (x, y). |

### Списки аргументов

Каждый вызов функции-члена и делегата включает список аргументов, в котором передаются фактические параметры или ссылки на переменные для параметров функции-члена. Синтаксис записи списка аргументов при вызове функции-члена зависит от категории функции-члена.

* Для конструкторов экземпляров, методов, индексаторов и делегатов аргументы указываются в виде списка аргументов, как показано ниже. Для индексаторов при вызове метода доступа set список аргументов дополнительно включает выражение, указанное в качестве правого операнда оператора присваивания.
* Для свойств список аргументов пуст при вызове метода доступа get и содержит выражение, указанное в качестве правого операнда оператора присваивания, при вызове метода доступа set.
* Для событий список аргументов состоит из выражения, указанного в качестве правого операнда оператора += или -=.
* В пользовательских операторах список аргументов состоит из одного операнда для унарного оператора и из двух операторов для бинарного оператора.

Аргументы свойств (§10.7), событий (§10.8) и пользовательских операторов (§10.10) всегда передаются как параметры значений (§10.6.1.1). Аргументы индексаторов (§10.9) всегда передаются как параметры значений (§10.6.1.1) или массивы параметров (§10.6.1.4). Для этих категорий функций-членов параметры ссылки и параметры вывода не поддерживаются.

Аргументы при вызове конструктора экземпляра, метода, индексатора или делегата указываются в виде списка аргументов:

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

Список аргументов содержит один или несколько аргументов, разделенных запятыми. Каждый аргумент состоит из необязательного имени аргумента, за которым следует значение аргумента. На аргумент с именем аргумента ссылаются, как на именованный аргумент, в то время, как аргумент без имени аргумента является позиционным аргументом. Для позиционного аргумента является его местонахождение после именованного аргумента в списке аргументов.

Значение аргумента может иметь одну из следующих форм.

* Выражение, указывающее, что аргумент передается в виде параметра значения (§10.6.1.1).
* Ключевое слово ref, за которым следует ссылка на переменную (§5.4), указывающее, что аргумент передается в виде параметра ссылки (§10.6.1.2). Переменная должна быть определенно назначенной (§5.3) до того, как ее можно будет передать в качестве параметра по ссылке. Ключевое слово out, за которым следует ссылка на переменную (§5.4), указывающее, что аргумент передается в виде параметра вывода (§10.6.1.3). После вызова функции-члена, в котором переменная передается в виде параметра вывода, переменная считается явно присвоенной (§5.3).

#### Соответствующие параметры

Для каждого аргумента в списке аргументов внутри члена функции или вызываемого делегата должен присутствовать соответствующий параметр.

Список параметров, использующийся в следующих случаях, определяется следующим образом:

* Для виртуальных методов, а также индексаторов, определенных в классах, список параметров выбирается из наиболее подходящего объявления или переопределения члена функции, начиная со статического типа получателя и выполняя поиск через его базовые классы.
* Для методов и индексаторов интерфейса список параметров выбирается из наиболее подходящего определения члена, начиная с типа интерфейса и выполняя поиск через базовые интерфейсы. Если список уникальных параметров не найден, создается список параметров с недоступными именами и без необязательных параметров; таким образом, при вызовах не смогут использоваться именованные параметры и пропускаться необязательные аргументы.
* Для разделяемых методов используется список параметров определяющего объявления разделяемого метода.
* Для всех остальных членов функций и делегатов существует только один список параметров, который и используется.

Положение аргумента или параметра определяется как число аргументов или параметров, предшествующих ему в списке аргументов или списке параметров.

Соответствующие параметры для аргументов членов функции устанавливаются следующим образом:

* Аргументы в списке аргументов конструкторов экземпляров, методов, индексаторов и делегатов:
* Позиционный аргумент, в котором фиксированный параметр встречается в той же позиции, что и в списке параметров, соответствует этому параметру.
* Позиционный аргумент члена функции, в котором массив параметров вызван в нормальной форме, соответствует массиву параметров, который должен находиться в этой же позиции в списке параметров.
* Позиционный аргумент члена функции, в котором массив параметров вызван в расширенной форме, а в соответствующей ему позиции в списке параметров нет фиксированного параметра, соответствует элементу в массиве параметров.
* Именованный аргумент соответствует параметру с тем же именем в списке параметров.
* Для индексаторов, при вызове метода доступа set выражение, указанное как правый операнд оператора назначения, соответствует неявному параметру value объявления метода доступа set.
* Для свойств при вызове метода доступа get аргументов нет. При вызове метода доступа set выражение, указанное как правый операнд оператора назначения, соответствует неявному параметру value объявления метода доступа set.
* Для пользовательских унарных операторов (включая преобразования) один операнд соответствует одному параметру объявления оператора.
* Для пользовательских бинарных операторов левый операнд соответствует первому параметру, а правый - второму параметру объявления оператора.

#### Вычисление списков аргументов во время выполнения

При обработке вызова функции-члена во время выполнения (§7.5.4) выражения или ссылки на переменные списка аргументов вычисляются слева направо в следующем порядке.

* Для параметра значения вычисляется выражение аргумента и выполняется неявное преобразование (§6.1) в соответствующий тип параметра. Результат становится исходным значением параметра значения при вызове функции-члена.
* Для параметра ссылки или вывода вычисляется ссылка на переменную и получившееся расположение в памяти становится расположением, которое представляется параметром при вызове функции-члена. Если ссылка на переменную, предоставленная в виде параметра ссылки или вывода, является элементом массива ссылочного типа, во время выполнения проверяется, совпадает ли тип элемента массива с типом параметра. В случае неудачного завершения проверки возникает исключение System.ArrayTypeMismatchException.

В методах, индексаторах и конструкторах экземпляров самый правый параметр может объявляться массивом параметров (§10.6.1.4). Такие функции-члены вызываются либо в нормальной форме, либо в расширенной в зависимости от того, какая из форм применима (§7.5.3.1):

* Когда функция-член с массивом параметров вызывается в нормальной форме, аргумент массива параметров должен быть одним выражением с типом, который можно неявно преобразовать (§6.1) в тип массива параметров. В этом случае использование массива параметров не отличается от использования параметра значения.
* Когда функция-член с массивом параметров вызывается в расширенной форме, при вызове необходимо указать любое число аргументов для массива параметров, где каждый аргумент представляет собой выражение с типом, который можно неявно преобразовать (§6.1) в тип элементов массива параметров. В этом случае при вызове создается экземпляр типа массива параметров с длиной, соответствующей числу аргументов, происходит инициализация элементов экземпляра массива с помощью указанных значений аргументов и в качестве фактического аргумента используется этот только что созданный экземпляр массива.

Выражения списка аргументов всегда вычисляются в порядке их записи. Таким образом, в примере

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++);  
 }  
}

производятся следующие выходные данные

x = 0, y = 1, z = 2  
x = 4, y = -1, z = 3

По правилам ковариации массива (§12.5) значение массива типа A[] может быть ссылкой на экземпляр массива типа B[], если существует неявное преобразование ссылочного типа из B в A. Согласно этим правилам, когда элемент массива ссылочного типа передается в качестве параметра ссылки или вывода, во время выполнения необходимо убедиться, что фактический тип элементов массива идентичен типу параметра. В этом примере

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  
 }  
}

второй вызов F приводит к исключению System.ArrayTypeMismatchException, поскольку фактический тип элемента для b — string, а не object.

Когда функция-член с массивом параметров вызывается в расширенной форме, вызов обрабатывается точно так же, как если бы для расширенных параметров было указано выражение создания массива с инициализатором массива (§7.6.10.4). Например, при объявлении

void F(int x, int y, params object[] args);

следующие вызовы расширенной формы метода

F(10, 20);  
F(10, 20, 30, 40);  
F(10, 20, 1, "hello", 3.0);

в точности соответствуют

F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});

В частности, обратите внимание, что когда в массиве параметров аргументы не передаются, создается пустой массив.

Когда аргументы в функции-члене с соответствующими необязательными параметрами пропускаются, выполняется неявная передача аргументов по умолчанию объявления функции-члена. Так как они всегда постоянны, их вычисление не повлияет на порядок вычисления остальных аргументов.

### Вывод типа

При вызове универсального метода без указания аргументов типа процесс вывода типа пытается определить аргументы типа для вызова. Наличие вывода типа позволяет использовать более удобный синтаксис для вызова универсального метода и позволяет разработчику избегать указания избыточных данных о типе. Например, при объявлении метода

class Chooser  
{  
 static Random rand = new Random();

public static T Choose<T>(T first, T second) {  
 return (rand.Next(2) == 0)? first: second;  
 }  
}

можно вызвать метод Choose без явного указания аргумента типа:

int i = Chooser.Choose(5, 213); // Calls Choose<int>

string s = Chooser.Choose("foo", "bar"); // Calls Choose<string>

С помощью вывода типа аргументы типа int и string определяются из аргументов метода.

Вывод типа происходит при обработке вызова метода во время компиляции (§7.6.5.1) до этапа разрешения перегрузки для вызова. Когда в вызове метода указывается конкретная группа методов и не указываются аргументы типа, вывод типа применяется к каждому универсальному методу в группе методов. Если вывод типа завершается успехом, то полученные аргументы типа используются для определения типов аргументов для последующего разрешения перегрузки. Если при разрешении перегрузки выбирается вызов универсального метода, то при вызове в качестве фактических аргументов типа будут использоваться полученные аргументы. Если вывод типа для конкретного метода завершается неудачно, то такой метод не участвует в разрешении перегрузки. Ошибка при выводе типа сама по себе не приводит к ошибке времени привязки. Однако она часто приводит к ошибке времени привязки, когда при разрешении перегрузки не удается найти применимые методы.

Если предоставленное число аргументов отличается от числа параметров в методе, то вывод типа немедленно завершается ошибкой. Иначе предположим, что у универсального метода имеется следующая сигнатура

Tr M<X1…Xn>(T1 x1 … Tm xm)

При вызове метода в виде M(E1 …Em) задачей вывода типа является найти уникальные аргументы типа S1…Sn для каждого параметра типа X1…Xn, чтобы вызов M<S1…Sn>(E1…Em) оказался действительным.

Во время процесса вывода каждый параметр типа Xi либо *фиксирован* с определенным типом Si, либо *нефиксирован* со связанным набором *границ.* Каждая граница представляет собой некоторый тип T. Изначально каждая переменная типа Xi является нефикисированной с пустым набором границ.

Вывод типа происходит поэтапно. На каждом этапе выполняется попытка вывести аргументы типа для большего числа переменных типа на основании результатов предыдущего этапа. На первом этапе выводятся начальные границы, на втором этапе переменные связываются с определенными типами и выводятся дополнительные границы. Второй этап может потребоваться повторить несколько раз.

*Обратите внимание:* вывод типа происходит не только при вызове универсального метода. Вывод типа для преобразования групп методов описывается в разделе §7.5.2.13, а поиск самого подходящего общего типа для набора выражений описывается в разделе §7.5.2.14.

#### Первый этап

Для каждого аргумента метода Ei:

* Если Ei является анонимной функцией, выполняется явный вывод типа параметра (§7.5.2.7) из Ei к Ti.
* В противном случае, если Ei имеет тип U, а xi — параметр значения, выполняется *вывод по нижней границе* от U до Ti.
* В противном случае, если Ei имеет тип U, а xi является параметром ref или out, выполняется точный вывод из U в Ti.
* В противном случае вывод для этого аргумента не выполняется.

#### Второй этап

Второй этап продолжается следующим образом.

* Фиксируются все нефиксированные переменные типа Xi, которые не зависят (§7.5.2.5) от каких-либо Xj (§7.5.2.10).
* Если таких переменных типа нет, то фиксируются все нефиксированные переменные типа Xi, для которых верно, что:
  + существует по крайней мере одна переменная типа Xj , которая зависит от Xi;
  + Xi имеет непустой набор границ.
* Если таких переменных типа не существует и все еще остаются нефиксированные переменные типа, то вывод типа завершается сбоем.
* Иначе, если нефиксированных переменных типа больше не существует, вывод типа завершается успешно.
* Иначе для всех аргументов Ei с соответствующим типом параметра Ti, где выходные типы (§7.5.2.4) содержат нефиксированные переменные типа Xj, а входные типы (§7.5.2.3) не содержат, выполняется вывод выходного типа (§7.5.2.6) от Ei к Ti. После этого второй этап повторяется.

#### Типы ввода

Если E является группой методов или анонимной функцией с неявным указанием типа, а T является типом делегата или типом дерева выражения, то все типы параметров T являются *типами ввода* E с типомT.

#### Типы вывода

Если E является группой методов или анонимной функцией, а T является типом делегата или типом дерева выражения, то типом возвращаемого значения T является *тип вывода* E *с типом* T.

#### Зависимость

Нефиксированная переменная типа Xi *непосредственно зависит от* нефиксированной переменной типа Xj, если некоторый аргумент Ek с типом Tk Xj оказывается во входном типе Ek с типом Tk, а Xi оказывается в выходном типе Ek с типом Tk.

Xj *зависит от* Xi, если Xj непосредственно зависит от Xi или если Xi непосредственно зависит от Xk и Xk зависит от Xj. Таким образом, отношение зависит от является транзитивным, но не является рефлексивным замыканием отношения непосредственно зависит от.

#### Вывод типа вывода

Вывод типа вывода выполняется из выражения E в тип T следующим образом:

* Если E является анонимной функцией с полученным типом возвращаемого значения U (§7.5.2.12), а T является типом делегата или типом дерева выражения с типом возвращаемого значения Tb, то выполняется вывод по нижней границе (§7.5.2.9) из U к Tb.
* Иначе, если E является группой методов, а T является типом делегата или типом дерева выражений с типами параметров T1…Tk и типом возвращаемого значения Tb и разрешение перегрузки E с типами T1…Tk дает один метод с типом возвращаемого значения U, то выполняется вывод по нижней границе из U к Tb.
* Иначе, если E является выражением с типом U, то выполняется вывод по нижней границе из U к T.
* Иначе вывод не производится.

#### Вывод явных типов параметров

Явный вывод типа параметра выполняется из выражения E к типу T следующим образом:

* Если E является анонимной функцией с явным указанием типа с типами параметров U1…Uk, а T является типом делегата или типом дерева выражения с типами параметров V1…Vk, то для каждого Ui выполняется точный вывод (§7.5.2.8) из Ui к соответствующему Vi.

#### Точный вывод

Точный вывод из типа U к типу V выполняется следующим образом.

* Если V является одним из нефиксированных Xi, то U добавляется к набору границ для Xi.
* Иначе, наборы V1…Vk  и U1…Uk определяются проверкой применимости одного из следующих случаев:
* V является типом массива V1[…] и U является типом массива U1[…] того же ранга.
* V является типом V1? и U является типом U1?.
* V является сформированным типом C<V1…Vk> and U — сформированным типом C<U1…Uk> .

Если применим хотя бы один из этих случаев, выполняется точный вывод из каждого Ui в соответствующий Vi.

* Иначе вывод не производится.

#### Вывод нижних границ

Вывод нижних границ из типа U в тип V выполняется следующим образом.

* Если V является одним из нефиксированных Xi, то U добавляется к набору нижних границ для Xi.
* Иначе, если V является типом V1? и U является типом U1?, то выполняется вывод по нижней границе из U1 к V1.
* Иначе, наборы U1…Uk и V1…Vk определяются проверкой применимости одного из следующих случаев:
* V является типом массива V1[…], а U — типом массива U1[…] (или типом параметра, эффективный базовый тип которого — U1[…]) того же ранга
* V — один из IEnumerable<V1>, ICollection<V1> или IList<V1>, а U — тип одномерного массива U1[] (или параметр типа, эффективный базовый тип которого — U1[]).
* V — сформированный класс, структура, интерфейс или тип делегата C<V1…Vk>; также существует уникальный тип C<U1…Uk> , в котором U (или, если U является параметром типа, его эффективный базовый класс или любой член его набора эффективных интерфейсов) идентичен, наследует от (напрямую или косвенно) или реализует (напрямую или косвенно) C<U1…Uk>..

(Ограничение "уникальности" означает, что в случае interface C<T>{} class U: C<X>, C<Y>{} во время вывода из U в C<T> вывод произведен не будет, так как U1 может быть X или Y.)

Если применим хотя бы один из этих случаев, выполняется точный вывод из каждого Ui в соответствующий Vi следующим образом:

* Если неизвестно, что Ui  является ссылочным типом, выполняется *точный вывод*.
* Иначе, если U — тип массива, выполняется *вывод по нижней границе*.
* Иначе, если V является C<V1…Vk>, вывод зависит от i-го параметра типа C:
* Если он ковариантен, выполняется *вывод по нижней границе*.
* Если он контрвариантен, выполняется *вывод по верхней границе*.
* Если он инвариантен, выполняется *точный вывод*.
* Иначе вывод не производится.

#### Вывод по верхним границам

Вывод верхних границ из типа U в тип V выполняется следующим образом.

* Если V является одним из нефиксированных Xi, то U добавляется к набору верхних границ для Xi.
* Иначе, наборы V1…Vk  и U1…Ukопределяются проверкой применимости одного из следующих случаев:
* U является типом массива U1[…]и V является типом массива V1[…] того же ранга.
* U — один из IEnumerable<Ue>, ICollection<Ue> или IList<Ue>, а V — тип одномерного массива Ve[].
* U является типом U1? и V является типом V1?.
* U — сформированный класс, структура, интерфейс или тип делегата C<U1…Uk>, а V — класс, структура, интерфейс или тип делегата, идентичный, наследующий от (напрямую или косвенно), или реализующий (напрямую или косвенно) уникальный тип C<V1…Vk>.

(Ограничение "уникальности" означает, что в случае interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{} при выводе из C<U1> в V<Q> вывод производиться не будет. Вывод не выполняется из U1 в X<Q> или Y<Q>.)

Если применим хотя бы один из этих случаев, выполняется точный вывод из каждого Ui в соответствующий Vi следующим образом:

* Если неизвестно, что Ui  является ссылочным типом, выполняется *точный вывод*.
* Иначе, если V является типом массива, выполняется *вывод по верхней границе*.
* Иначе, если U является C<U1…Uk>, вывод зависит от i-го параметра типа C:
* Если он ковариантен, выполняется *вывод по верхней границе*.
* Если он контрвариантен, выполняется *вывод по нижней границе*.
* Если он инвариантен, выполняется *точный вывод*.
* Иначе вывод не производится.

#### Фиксирование

Нефиксированная переменная типа Xi с набором границ фиксируется следующим образом.

* Набор *типов кандидатов* Uj изначально формируется как набор всех типов в наборе границ для Xi.
* Затем по очереди проверяется каждая граница для Xi: для каждой точной границы U для Xi все типы Uj, не являющиеся идентичными U, удаляются из набора кандидатов. Для каждой нижней границы U для Xi из набора кандидатов удаляются все типы Uj, в которые *не* существует неявного преобразования из U. Для каждой верхней границы U для Xi из набора кандидатов удаляются все типы Uj, из которых *не* существует неявного преобразования в U.
* Если среди оставшихся типов-кандидатов Uj имеется уникальный тип V, из которого существует неявное преобразование во все остальные типы-кандидаты, то Xi фиксируется в V.
* Иначе вывод типа завершается сбоем.

#### Выведенный тип возвращаемого значения

Выведенный тип возвращаемого значения анонимной функции F используется при выводе типа и разрешении перегрузки. Выведенный тип возвращаемого значения можно определить только для анонимной функции со всеми известными типами параметров, которые либо заданы явно и переданы через преобразование анонимной функции, либо выведены во время вывода типа для включающего вызова универсального метода.

Выведенный тип возвращаемого значения определяется следующим образом.

* Если телом F является выражение, имеющее тип, то результирующий выведенный тип F совпадет с типом этого выражения.
* Если телом F является блок, а самым подходящим общим типом для набора выражений в операторах return блока является T (§7.5.2.14), то выведенным типом результата F будет T.
* Иначе тип результата для F вывести нельзя.

Выведенный тип возвращаемого значения определяется следующим образом.

* Если F — асинхронная функция и тело F — выражение, классифицированное как Nothing (§7.1), или блок операторов, в котором операторы return не имеют выражений, выведенным типом возвращаемого значения является System.Threading.Tasks.Task
* Если функция F является синхронной и имеет выведенный тип возвращаемого значения T, выведенным типом возвращаемого значения будет System.Threading.Tasks.Task<T>.
* Если функция F не является синхронной и имеет выведенный тип возвращаемого значения T, выведенным типом возвращаемого значения будет T.
* Иначе тип возвращаемого значения для F вывести нельзя.

В качестве примера вывода типа, включающего анонимную функцию, рассмотрим метод расширения Select, объявленный в классе System.Linq.Enumerable:

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);  
 }  
 }  
}

Предположим, что пространство имен System.Linq было импортировано с помощью предложения using и свойство Name класса Customer имеет тип string. Тогда для выбора имен списка клиентов можно использовать метод Select:

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

Вызов метода расширения (§7.6.5.2) для Select обрабатывается путем перезаписи вызова статического метода:

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

Поскольку аргументы типа не были указаны явно, для получения аргументов типа используется вывод типа. Сначала аргумент customers связывается с параметром source, и для T выводится тип Customer. Затем, используя описанный выше процесс вывода типа анонимной функции, c присваивается тип Customer, а выражение c.Name связывается с типом возвращаемого значения параметра selector, в результате чего для S выводится тип string. Таким образом, этот вызов эквивалентен

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

а результат имеет тип IEnumerable<string>.

В следующем примере демонстрируется, как вывод типа анонимной функции обеспечивает «перетекание» данных о типе между аргументами в вызове универсального метода. Если имеется метод:

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

Вывод типа при вызове:

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

продолжается следующим образом: сначала аргумент "1:15:30" связывается с параметром value и для X выводится тип string. Затем параметру первой анонимной функции s присваивается выведенный тип string, а выражение TimeSpan.Parse(s) связывается с типом возвращаемого значения f1, в результате чего для Y выводится тип System.TimeSpan. И наконец, параметру второй анонимной функции t присваивается выведенный тип System.TimeSpan, а выражение t.TotalSeconds связывается с типом возвращаемого значения f2, в результате чего для Z выводится тип double. Таким образом, результат вызова имеет тип double.

#### Вывод типа при преобразовании групп методов

Аналогично вызовам универсальных методов вывод типа также должен применяться, когда группа методов M, включающая универсальный метод, преобразуется в данный тип делегата D (§6.6). Если имеется метод

Tr M<X1…Xn>(T1 x1 … Tm xm)

и группа методов M назначается типу делегата D, то задачей вывода типа является поиск аргументов типа S1…Sn, чтобы выражение:

M<S1…Sn>

оказалось совместимым (§15.1) с D.

В отличие от алгоритма вывода для вызовов универсальных методов, в данном случае имеются только типы аргументов, но не выражения аргументов. В частности, нет анонимных функций, следовательно, нет необходимости в нескольких этапах вывода.

Вместо этого все Xi считаются нефиксированными, и выполняется вывод по нижней границе из каждого типа аргумента Uj для D в соответствующий тип параметра Tj для M. Если для какого-либо Xi границы не были найдены, вывод типа завершается сбоем. Иначе все Xi фиксируются в соответствующие Si, которые являются результатом вывода типа.

#### Поиск наиболее подходящего общего типа для набора выражений

В некоторых случаях для набора выражений необходимо вывести общий тип. В частности, типы элементов для массивов с неявным указанием типа и типы возвращаемых значений анонимных функций с телом в виде блока находятся следующим образом.

При наличии набора выражений E1…Em такой вывод должен быть эквивалентен вызову метода

Tr M<X>(X x1 … X xm)

с Ei в качестве аргументов.

Точнее, вывод начинается с нефиксированной переменной типа X. Вывод выходных типов выполняется из каждого Ei в тип X. Наконец, X фиксируется и, в случае успеха, получившийся тип S становится наиболее подходящим общим типом для выражений. Если S не существует, выражения не имеют наиболее общего типа.

### Разрешение перегрузки

Разрешение перегрузки — это механизм времени привязки для выбора наиболее подходящего члена-функции для вызова по предоставленному списку аргументов и набору кандидатов членов-функций. Разрешение перегрузки позволяет выбрать член-функцию для вызова в следующих отдельных контекстах в C#:

* вызов метода, указанного в выражении вызова (§7.6.5.1);
* вызов конструктора экземпляра, указанного в выражении создания объекта (§7.6.10.1);
* вызов метода доступа к индексатору с помощью доступа к элементу (§7.6.6);
* вызов стандартного или пользовательского оператора, на который есть ссылка в выражении (§7.3.3 и §7.3.4).

Для каждого из этих контекстов набор кандидатов функций-членов и список аргументов определяются по-своему, как подробно описывается в разделах выше. Например, набор кандидатов для вызова метода не включает методы, помеченные с помощью ключевого слова override (§7.4), кроме того в список кандидатов не попадают методы базового класса, если применим любой из методов производного класса (§7.6.5.1).

После определения кандидатов функций-членов и списка аргументов выбор наиболее подходящей функции-члена осуществляется одинаковым образом для всех случаев.

* При наличии набора применимых кандидатов функций-членов определяется наиболее подходящая функция-член. Если набор содержит только одну функцию-член, то наиболее подходящей функцией-членом является эта функция. Иначе наиболее подходящей функцией-членом является функция-член, более всего подходящая по списку аргументов в соответствии с правилами из раздела §7.5.3.2. Если не удалось найти одну функцию-член, более подходящую, чем остальные, то вызов функции-члена оказывается неоднозначным и возникает ошибка времени привязки.

В следующих разделах определяется значение терминов применимая функция-член и более подходящая функция-член.

#### Применимая функция-член

Функция-член называется применимой функцией-членом в соответствии со списком аргументов A, когда выполняются все следующие условия.

* Каждый аргумент в A соответствует параметру в объявлении функции члена, как описано в §7.5.1.1, и любой параметр, которому не соответствует аргумент, является необязательным параметром.
* Для каждого аргумента в A режим передачи параметра для аргумента (то есть по значению, с помощью ключевых слов ref или out) совпадает с режимом передачи соответствующего параметра и
* для параметра значения или массива параметров существует неявное преобразование (§6.1) из аргумента в тип соответствующего параметра или
* для параметров, передаваемых с помощью ref или out, тип аргумента идентичен типу соответствующего параметра. В конечном счете параметры ref или out представляют собой псевдонимы для передаваемых аргументов.

Для функции-члена, включающей массив параметров, если функция-член применима в соответствии с указанными правилами, то она называется применимой в нормальной форме. Если функция-член, включающая массив параметров, не применима в нормальной форме, то она может быть применима в расширенной форме.

* Расширенная форма создается путем замены массива параметров в объявлении функции-члена на ноль или другое число параметров с типом элементов массива параметров, так чтобы число аргументов в списке аргументов A совпадало с общим числом параметров. Если в списке A меньше аргументов, чем число фиксированных параметров в объявлении функции-члена, то создать расширенную форму функции-члена нельзя и поэтому она неприменима.
* Иначе расширенная форма применима, если для каждого аргумента в списке A режим передачи параметра для аргумента идентичен режиму передачи соответствующего параметра и
* для фиксированного параметра значения или параметра значения, созданного при расширении, существует неявное преобразование (§6.1) из типа аргумента в тип соответствующего параметра или
* для параметров, передаваемых с помощью ref или out, тип аргумента идентичен типу соответствующего параметра.

#### Более подходящая функция-член

Для целей определения наиболее подходящей функции-члена формируется укороченный список аргументов А, содержащий только сами выражения аргументов в том порядке, в каком они присутствуют в исходном списке аргументов.

Списки параметров для каждого кандидата функции-члена формируется следующим способом:

* Расширенная форма используется в случае, если функция-член была применима только в расширенной форме.
* Необязательные параметры без соответствующих аргументов удаляются из списка параметров
* Порядок параметров изменяется таким образом, что они встречаются в той же позиции, что и соответствующий аргумент в списке аргументов.

При наличии списка аргументов A с набором выражений аргументов { E1, E2, ..., EN } и двух применимых функций-членов MP и MQ с типами параметров { P1, P2, ..., PN } и { Q1, Q2, ..., QN } MP считается более подходящей функцией-членом, чем MQ, если

* для каждого аргумента неявное преобразование из EX в QX не лучше неявного преобразования из EX в PX и
* по крайней мере для одного аргумента преобразование из EX в PX лучше преобразования из EX в QX.

При проведении этой оценки если MP или MQ применима в расширенной форме, то PX или QX относится к параметру в расширенной форме списка параметров.

Если последовательности типов параметров {P1, P2, …, PN} и {Q1, Q2, …, QN} идентичны (т. е. для каждого Pi имеется тождественное преобразование в соответствующий Qi), то для определения более подходящей функции-члена применяются следующие правила разрешения.

* Если MP не является универсальным методом, а MQ — универсальный метод, то MP является более подходящей, чем MQ.
* Иначе, если MP применима в нормальной форме, а MQ имеет массив params и применима только в расширенной форме, то MP является более подходящей, чем MQ.
* В противном случае, если у MP больше объявленных параметров, чем у MQ, то MP является более подходящей, чем MQ. Это может происходить, когда у обеих функций-членов есть массивы params, применимые только в расширенных формах.
* Иначе, если у всех параметров MP есть соответствующий аргумент, тогда как аргументы по умолчанию должны быть заменены на по меньшей мере один необязательный параметр в MQ, то MP является более подходящей, чем MQ.
* В противном случае, если у MP более конкретные типы параметров, чем у MQ, то MP является более подходящей, чем MQ. Пусть наборы {R1, R2, …, RN} и {S1, S2, …, SN} представляют нерасширенные типы параметров без экземпляров для MP и MQ. Типы параметров MP являются более конкретными, чем у MQ, если для любого параметра RX является не менее конкретным, чем SX, и по крайней мере один параметр RX является более конкретным, чем SX.
* Параметр типа является менее конкретным, чем не параметр типа.
* Один сформированный тип является более конкретным, чем другой (с таким же числом аргументов типа), если у первого типа по крайней мере один аргумент типа является более конкретным по сравнению с соответствующим аргументом типа второго типа и нет аргументов типа менее конкретных по сравнению с соответствующими аргументами типа другого сформированного типа.
* Тип массива является более конкретным, чем другой тип массива (с таким же числом измерений), если тип элементов первого является более конкретным, чем тип элементов второго.
* Иначе, если один член не является оператором с нулификацией, а другой является, то оператор без нулификации считается более подходящим.
* Иначе определить более подходящую функцию-член нельзя.

#### Лучшее преобразование из выражения

При наличии неявного преобразования C1, которое переводит выражение E в тип T1, и неявного преобразования C2, которое переводит выражение E в тип T2, C1 является лучшим преобразованием, чем C2, если верно хотя бы одно из следующих утверждений:

* E имеет тип S, также существует преобразование идентификации из S в T1, но не из S в T2.
* E не является анонимной функцией, а T1 является более подходящей целью преобразования, чем T2 (§7.5.3.5).
* E является анонимной функцией, T1 является либо типом делегата D1, либо типом дерева выражения Expression<D1>, T2 является либо типом делегата D2, либо типом дерева выражения Expression<D2>, и верно одно из следующих утверждений:
* D1 является более подходящей целью преобразования, чем D2.
* D1 и D2 имеют идентичные списки параметров, и верно одно из следующих утверждений:
* D1 имеет тип возвращаемого значения Y1, а D2 — тип возвращаемого значения Y2, выведенный тип возвращаемого значения X существует для E в контексте данного списка параметров (§7.5.2.12), и преобразование из X в Y1 является более подходящим, чем преобразование из X в Y2.
* E является асинхронным выражением, D1 имеет тип возвращаемого значения Task<Y1>, а D2 имеет тип возвращаемого значения Task<Y2>, выведенный тип возвращаемого значения Task<X> существует для E в контексте данного списка параметров (§7.5.2.12), и преобразование из X в Y1 является более подходящим, чем преобразование из X в Y2.
* D1 имеет тип возвращаемого значения Y, а D2 возвращает значение void

#### Лучшее преобразование из типа

При наличии преобразования C1, которое переводит тип S в тип T1, и преобразования C2, которое переводит тип S в тип T2, C1 является лучшим преобразованием, чем C2, если верно хотя бы одно из следующих утверждений:

* Существует тождественное преобразование из S в T1, и не существует преобразования из S в T2.
* T1 является лучшей целью для преобразования, чем T2 (§7.5.3.5).

#### Лучшая цель для преобразования

В случае, если даны два различных типа T1 и Т2, Т1 является лучшей целью для преобразования, чем Т2, если верно одно из следующих утверждений:

* Существует неявное преобразование из T1 в T2, и нет неявного преобразования из T2 в T1.
* T1 является целым типом со знаком, а T2 является целым типом без знака. А именно:
* T1 имеет тип sbyte, а T2 имеет тип byte, ushort, uint или ulong
* T1 имеет тип short, а T2 имеет тип ushort, uint или ulong.
* T1 имеет тип int, а T2 имеет тип uint или ulong.
* T1 равно long и T2 равно ulong.

#### Перегрузка в универсальных классах

Несмотря на то что сигнатуры при объявлении должны быть уникальными, при замене аргументов типа могут создаваться идентичные сигнатуры. В таких случаях конечные правила разрешения перегрузки, описанные выше, позволят выбрать конкретный член.

В следующих примерах показываются допустимые и недопустимые в соответствии с этим правилом перегрузки:

interface I1<T> {...}

interface I2<T> {...}

class G1<U>  
{  
 int F1(U u); // Overload resulotion 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);  
}

### Проверка динамического разрешения перегрузки во время компиляции

Для основных динамически связанных операций набор возможных кандидатов для разрешения во время компиляции неизвестен. В некоторых случаях, однако, набор кандидатов известен во время компиляции:

* Вызовы статических методов с динамическими аргументами
* Вызовы методов экземпляра, в случаях, когда получатель не является динамическим выражением
* Вызовы индексатора, в случаях, когда получатель не является динамическим выражением
* Вызовы конструктора с динамическими аргументами

В этих случаях для каждого кандидата выполняется ограниченная проверка времени компиляции, выявляющая потенциальную применимость во время выполнения. Эта проверка состоит из следующих этапов:

* Частичный вывод типа: Любой аргумент типа, не зависящий напрямую или косвенно от аргумента типа dynamic, выводится с использованием правил §7.5.2. Остальные аргументы типа неизвестны.
* Частичная проверка применимости: Проверяется применимость в соответствии с §7.5.3.1, но параметры, тип которых неизвестен, пропускаются.

Если кандидатов, прошедших эту проверку, нет, происходит ошибка времени компиляции.

### Вызов функции-члена

В этом разделе описывается процесс, который происходит во время выполнения при вызове определенной функции-члена. Предполагается, что во время компиляции уже была определена конкретная вызываемая функция-член, вероятно, с помощью разрешения перегрузки для набора кандидатов функций-членов.

В целях описания процесса вызова функции-члены разделены на две категории.

* Статические функции-члены. Это конструкторы экземпляров, статические методы, методы доступа к статическим свойствам и пользовательские операторы. Статические функции-члены всегда являются невиртуальными.
* Функции-члены экземпляра. Это методы экземпляра, методы доступа к свойствам и индексаторам. Функции-члены экземпляра могут быть либо виртуальными, либо невиртуальными и всегда вызываются для конкретного экземпляра. Экземпляр вычисляется по выражению экземпляра и оказывается доступен внутри функции-члена с помощью оператора this (§7.6.7).

Обработка вызова функции-члена во время выполнения состоит из следующих этапов, где M —это функция-член, а E — это выражение экземпляра (когда M является членом экземпляра).

* Если M является статической функцией-членом
* Список аргументов вычисляется, как описано в разделе §7.5.1.
* Вызывается M.
* Если M является функцией-членом экземпляра, объявленной в типе значения:
* Вычисляется E. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Если E не является переменной, то создается временная локальная переменная типа E и этой переменной присваивается значение E. После этого E становится ссылкой на эту временную локальную переменную. Временная переменная доступна с помощью оператора this в M, но не другим способом. Таким образом, вызывающий может наблюдать изменения, проводимые в M для this, только когда E является настоящей переменной.
* Список аргументов вычисляется, как описано в разделе §7.5.1.
* Вызывается M. Переменная, на которую ссылается E, становится переменной, на которую ссылается this.
* Если M является функцией-членом экземпляра, объявленной в ссылочном типе:
* Вычисляется E. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Список аргументов вычисляется, как описано в разделе §7.5.1.
* Если E имеет тип значения, выполняется преобразование упаковки (§4.3.1) для преобразования E в тип object. На следующих этапах считается, что E имеет тип object. В данном случае M может быть только членом System.Object.
* Выполняется проверка допустимости значения E. Если E имеет значение null, вызывается исключение System.NullReferenceException и дальнейшие действия не выполняются.
* Определяется реализация вызываемой функции-члена.
* Если типом E во время компиляции является интерфейс, то вызываемой функцией-членом является реализация M, предоставленная типом времени выполнения экземпляра, на который ссылается E. Эта функция-член определяется применением правил сопоставления интерфейса (§13.4.4), чтобы определить реализацию M, предоставленную типом времени выполнения экземпляра, на который ссылается E.
* Иначе, если M является виртуальной функцией-членом, вызываемая функция-член является реализацией M, предоставленной типом времени выполнения экземпляра, на который ссылается E. Эта функция-член определяется применением правил определения самой производной реализации (§10.6.3) M в соответствии с типом времени выполнения экземпляра, на который ссылается E.
* Иначе M является невиртуальной функцией-членом, и вызываемой функцией-членом является сама M.
* Вызывается реализация функции-члена, определенная на этапе выше. Объект, на который ссылается E, становится объектом, на который ссылается this.

#### Вызов в упакованных экземплярах

Функция-член, реализованная в типе значения может вызываться через упакованный экземпляр этого типа значения в следующих ситуациях.

* Когда функция-член является перегрузкой override метода, унаследованного от типа object, и вызывается через выражение экземпляра типа object.
* Когда функция-член является реализацией функции-члена экземпляра и вызывается через выражение экземпляра типа интерфейса.
* Когда функция-член вызывается через делегат.

В таких ситуациях считается, что упакованный экземпляр содержит переменную с типом значения и эта переменная становится переменной, на которую ссылается оператор this в вызове функции-члена. В частности, это означает, что при вызове функции-члена в упакованном экземпляре она может изменять значение, хранящееся в упакованном экземпляре.

## Первичные выражения

Первичные выражения включают самые простые формы выражений.

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

Первичные выражения разделяются на выражения создания массива и первичные выражения создания не массива. Такое выделение выражений создания массива, вместо рассмотрения их вместе с другими простыми видами выражений, позволяет в грамматике запретить потенциально неоднозначный код, например:

object o = new int[3][1];

который в противном случае интерпретировался бы как

object o = (new int[3])[1];

### Литералы

Первичное выражение, состоящее из литерала (§2.4.4), считается значением.

### Простые имена

Простое имя состоит из идентификатора, за которым может следовать список аргументов типа:

simple-name:  
identifier type-argument-listopt

Простое имя имеет вид I или одну из форм I<A1, ..., AK>, где I — один идентификатор и <A1, ..., AK> — дополнительный список аргументов типа. Если список аргументов типа не указан, K считается равным нулю. Простое имя вычисляется и классифицируется следующим образом.

* Если K равно нулю и простое имя находится внутри блока и область объявления локальных переменных блока (или включающего блока) (§3.3) содержит локальную переменную, параметр или ограничение с именем I, то простое имя относится к этой переменной, параметру или константе и считается переменной или значением.
* Если K равно нулю и простое имя находится внутри тела объявления универсального метода и если это объявление включает параметр типа с именем I, то простое имя относится к этому параметру типа.
* Иначе для каждого типа экземпляра T (§10.3.1), начиная от типа экземпляра в объявлении непосредственного включающего типа и до типа экземпляра в объявлении каждого включающего класса и структуры (если есть)
* Если K равно нулю и объявление T включает параметр типа с именем I, то простое имя относится к этому параметру типа.
* Иначе, если при поиске члена I (§7.4) в T с аргументами типа K получены результаты:
* Если T имеет тип экземпляра непосредственного включающего класса или структуры и при поиске находится один или несколько методов, то результатом является группа методов со связанным выражением экземпляра this. Если указан список аргументов типа, он используется при вызове универсального метода (§7.6.5.1).
* Иначе, если T имеет тип экземпляра непосредственного включающего класса или структуры и при поиске находится член экземпляра и если внутри блока конструктора экземпляра, метода экземпляра или метода доступа к экземпляру находится ссылка, то результатом является метод доступа к члену (§7.6.4) в виде this.I. Это происходит, только когда K равно нулю.
* Иначе результатом является метод доступа к члену (§7.6.4) в виде T.I или T.I<A1, ...,. AK>. В таком случае если простое имя будет относиться к члену экземпляра, это будет вызывать ошибку времени компиляции.
* Иначе для каждого пространства имен N, начиная с пространства имен, в котором находится простое имя, и заканчивая всеми включающими пространствами имен (если есть) и глобальным пространством имен, пока не будет найдена сущность, будут выполняться следующие действия.
* Если K равно нулю и I является именем пространства имен в N, то:
* Если местонахождение простого имени включено в объявление пространства имен для N и объявление пространства имен содержит директиву extern alias или директиву using alias, которая связывает имя I с пространством имен или типом, тогда простое имя неоднозначно и возникает ошибка времени компиляции.
* Иначе простое имя относится к пространству имен I в N.
* Иначе, если N содержит доступный тип с именем I и параметрами типа K, то:
* Если K равно нулю и местонахождение простого имени включено в объявление пространства имен для N и объявление пространства имен содержит директиву extern alias или директиву using alias, которая связывает имя I с пространством имен или типом, тогда простое имя неоднозначно и возникает ошибка времени компиляции.
* Иначе имя пространства имен или типа относится к типу, сформированному с данными аргументами типа.
* Иначе, если местонахождение простого имени включено в объявление пространства имен для N, то:
* Если K равно нулю и объявление пространства имен содержит директиву extern alias или директиву using alias, которая связывает имя I с импортированным пространством имен или типом, тогда простое имя относится к этому пространству имен или типу.
* Иначе, если пространства имен, импортированные с помощью директив using namespace в объявлении пространства имен, содержат ровно один тип с именем I и параметрами типа K, тогда простое имя относится к этому типу, сформированному с данными аргументами типа.
* Иначе, если пространства имен, импортированные с помощью директив using namespace в объявлении пространства имен, содержат более одного типа с именем I и параметрами типа K, тогда простое имя неоднозначно и возникает ошибка.

Обратите внимание, что этот этап полностью повторяет соответствующий этап в обработке имени пространства имен или типа (§3.8).

* Иначе простое имя не определено и возникает ошибка времени компиляции.

#### Инвариантность значения в блоках

Для каждого вхождения данного идентификатора в качестве полного простого имени (без списка аргументов типа) в выражении или деклараторе внутри области объявления локальных переменных (§3.3), непосредственно включающей это вхождение, каждое вхождение того же самого идентификатора в качестве полного простого имени в выражении или декларатора должно ссылаться на одну и ту же сущность. Это правило позволяет гарантировать, что внутри определенного блока, блока switch, а также операторов for, foreach и using имя всегда имеет одинаковое значение.

Пример:

class Test  
{  
 double x;

void F(bool b) {  
 x = 1.0;  
 if (b) {  
 int x;  
 x = 1;  
 }  
 }  
}

возникнет ошибка времени компиляции, потому что x относится к другим сущностям во внешнем блоке (часть которого включает вложенный блок в операторе if). Наоборот, в примере

class Test  
{  
 double x;

void F(bool b) {  
 if (b) {  
 x = 1.0;  
 }  
 else {  
 int x;  
 x = 1;  
 }  
 }  
}

ошибки не возникает, поскольку имя x не используется во внешнем блоке.

Обратите внимание, что правило инвариантности значения применяется только к простым именам. Один идентификатор вполне может иметь одно значение в виде простого имени и другое значение в виде правого операнда в методе доступа к члену (§7.6.4). Пример:

struct Point  
{  
 int x, y;

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

В примере выше демонстрируется общий шаблон использования имен полей в качестве имен параметров в конструкторе экземпляра. В примере простые имена x и y относятся к параметрам, но это не мешает выражениям доступа к членам this.x и this.y иметь доступ к полям.

### Выражения со скобками

Выражение со скобками состоит из выражения, заключенного в скобки.

parenthesized-expression:  
( expression )

Выражение со скобками вычисляется путем вычисления выражения внутри скобок. Если выражение внутри скобок обозначает пространство имен или тип, возникает ошибка времени компиляции. Иначе результатом выражения со скобками является результат вычисления содержащегося выражения.

### Доступ к члену

Доступ к члену состоит из первичного выражения, стандартного типа или уточненного члена псевдонима, за которым следует точка «.» и идентификатор. В конце может находиться список аргументов типа.

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

Создание уточненного члена псевдонима определяется в разделе §9.7.

Доступ к члену имеет вид E.I или одну из форм E.I<A1, ..., AK>, где E — первичное выражение, I — один идентификатор, а <A1, ..., AK> — необязательный список аргументов типа. Если список аргументов типа не указан, K считается равным нулю.

Доступ к члену с первичным выражением типа dynamic привязывается динамически (§7.2.2). В этом случае компилятор классифицирует доступ к члену как доступ к свойству типа dynamic. Представленные ниже правила определения значения доступа к члену затем применяются во время выполнения, используя тип времени выполнения, а не тип времени компиляции первичного выражения. Если класс времени выполнения ведет к группе методов, то метод доступа к члену должен являться первичным выражением для выражения вызова.

Результат метода доступа к члену вычисляется и классифицируется следующим образом.

* Если K равно нулю, E является пространством имен и E содержит вложенное пространство имен I, то результатом является это пространство имен.
* Иначе, если E является пространством имен, E содержит доступный тип с именем I и параметрами типа K, то результатом является этот тип, сформированный с указанными аргументами типа.
* Если E является стандартным типом или первичным выражением, которое классифицируется как тип, и если E не является параметром типа и если при поиске члена I (§7.4) в E с параметрами типа K было найдено соответствие, то E.I вычисляется и классифицируется следующим образом.
* Если I обозначает тип, то результатом является этот тип, сформированный с указанными аргументами типа.
* Если I обозначает один или несколько методов, то результатом является группа методов без связанного выражения экземпляра. Если указан список аргументов типа, он используется при вызове универсального метода (§7.6.5.1).
* Если I обозначает свойство типа static, то результатом является метод доступа к свойству без связанного выражения экземпляра.
* Если I определяет поле static:
* Если поле предназначено только для чтения (readonly) и вне статического конструктора класса или структуры, в которой объявляется поле, имеется ссылка, то результатом является значение, а именно значение статического поля I в E.
* Иначе результатом является переменная, а именно статическое поле I в E.
* Если I определяет событие static:
* Если внутри класса или структуры, в которой объявляется событие, имеется ссылка и это событие было объявлено без объявлений метода доступа к событию (§10.8), то E.I обрабатывается точно так же, как если бы I было статическим полем.
* Иначе результатом является метод доступа к событию без связанного выражения экземпляра.
* Если I обозначает константу, то результатом является значение, а именно значение этой константы.
* Если I обозначает член перечисления, то результатом является значение, а именно значение этого члена перечисления.
* Иначе E.I является недопустимой ссылкой на член, и возникает ошибка времени компиляции.
* Если E является методом доступа к свойству или индексатору, переменной или значением с типом T и при поиске члена I (§7.4) в T с аргументами типа K было найдено соответствие, то E.I вычисляется и классифицируется следующим образом.
* Во-первых, если E является свойством или методом доступа к индексатору, то происходит получение значения свойства или метода доступа к индексатору (§7.1.1) и класс E меняется на значение.
* Если I обозначает один или несколько методов, то результатом является группа методов со связанным выражением экземпляра E. Если указан список аргументов типа, он используется при вызове универсального метода (§7.6.5.1).
* Если I обозначает свойство экземпляра, то результатом является метод доступа к свойству со связанным выражением экземпляра E.
* Если T является типом класса и I обозначает поле экземпляра этого типа класса, то
* Если значение E равно null, возникает исключение System.NullReferenceException.
* Иначе, если поле предназначено только для чтения (readonly) и вне статического конструктора класса или структуры, в которой объявляется поле, имеется ссылка, то результатом является значение, а именно значение поля I в объекте, на который ссылается E.
* Иначе результатом является переменная, а именно поле I в объекте, на который ссылается E.
* Если T является типом структуры и I обозначает поле экземпляра этого типа структуры, то
* Если E является значением или если поле предназначено только для чтения (readonly) и вне статического конструктора класса или структуры, в которой объявляется поле, имеется ссылка, то результатом является значение, а именно значение поля I в экземпляре структуры, на который ссылается E.
* Иначе результатом является переменная, а именно поле I в экземпляре структуры, предоставленном E.
* Если I обозначает событие экземпляра, то
* Если внутри класса или структуры, в которой объявляется событие, имеется ссылка, данное событие было объявлено без объявлений метода доступа к событию (§10.8) и ссылка не находится в левой части оператора += или -=, то E.I обрабатывается точно так же, как если бы I было полем экземпляра.
* Иначе результатом является метод доступа к событию со связанным выражением экземпляра E.
* Иначе выполняется попытка обработать E.I как вызов метода расширения (§7.6.5.2). Если такой вызов завершается сбоем, E.I является недопустимой ссылкой на член и возникает ошибка времени привязки.

#### Идентичные простые имена и имена типов

В методе доступа к члену в виде E.I, где E является простым идентификатором, а значение E в качестве простого имени (§7.6.2) является константой, полем, свойством, локальной переменной или параметром того же типа, что и значение E в качестве имени типа (§3.8), то допустимы оба возможных значения E. Два возможных значения E.I никогда не являются неоднозначными, поскольку I в обоих случаях обязательно должен быть членом типа E. Другими словами, это правило просто разрешает доступ к статическим членам и вложенным типам E, когда в противном случае возникла бы ошибка времени компиляции. Пример:

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  
 }  
}

Внутри класса A вхождения идентификатора Color, которые ссылаются на тип Color, подчеркиваются, а которые ссылаются на поле Color, не подчеркиваются.

#### Грамматическая неоднозначность

Порождения для простого имени (§7.6.2) и доступа к члену (§7.6.4) могут привести к появлению неоднозначности в грамматике выражений. Например, оператор

F(G<A,B>(7));

может интерпретироваться как вызов F с двумя аргументами: G < A и B > (7). Иначе он может интерпретироваться как вызов F с одним аргументом, который представляет собой вызов универсального метода G с двумя аргументами типа и одним обычным аргументом.

Если последовательность лексем (в контексте) можно разобрать как простое имя (§7.6.2), метод доступа к члену (§7.6.4) или метод доступа к указателю члена (§18.5.2), заканчивающийся списком аргументов типа (§4.4.1), то выполняется проверка лексемы, следующей непосредственно за закрывающей лексемой >. Если это одна из лексем

( ) ] } : ; , . ? == != | ^

то список аргументов типа остается в составе простого имени, доступа к члену или доступа к члену-указателю, а любой другой возможный разбор последовательности лексем игнорируется. Иначе список аргументов типа не считается частью простого имени, доступа к члену или доступа к члену-указателю, даже если не существует другого возможного разбора последовательности лексем. Обратите внимание, что эти правила не применяются при разборе списка аргументов типа в имени пространства имен или типа (§3.8). Оператор

F(G<A,B>(7));

в соответствии с этим правилом будет интерпретироваться как вызов F с одним аргументом, который представляет собой вызов универсального метода G с двумя аргументами типа и одним обычным аргументом. Каждый оператор

F(G < A, B > 7);  
F(G < A, B >> 7);

будет интерпретирован как вызов F с двумя аргументами. Оператор

x = F < A > +y;

будет интерпретирован как оператор «меньше чем», оператор «больше чем» и унарный оператор «плюс», как если бы выражение было записано в виде x = (F < A) > (+y), а не в виде простого имени со списком аргументов типа, за которым следует бинарный оператор «плюс». В операторе

x = y is C<T> + z;

лексемы C<T> интерпретируются как имя пространства имен или типа со списком аргументов типа.

### Выражения вызова

Выражение вызова используется для вызова метода.

invocation-expression:  
primary-expression ( argument-listopt )

Выражение вызова динамически привязано (§7.2.2), если верно по меньшей мере одно из следующих утверждений:

* Основное выражение имеет тип времени компиляции dynamic.
* По меньшей мере один аргумент необязательного списка аргументов имеет тип времени компиляции dynamic, и основное выражение не имеет типа делегата.

В этом случае компилятор классифицирует выражение вызова как значение типа dynamic. Представленные ниже правила определения значения выражения вызова затем применяются во время выполнения, используя тип времени выполнения вместо правил типа времени компиляции первичного выражения и аргументов, имеющих тип времени компиляции dynamic. Если основное выражение не имеет тип времени компиляции dynamic, то для вызова метода будет выполнена ограниченная проверка времени компиляции, как описано в §7.5.4.

Основное выражение выражения вызова должно быть группой методов или значением типа делегата. Если основное выражение является группой методов, то выражение вызова является вызовом метода (§7.6.5.1). Если основное выражение является значением типа делегата, то выражение вызова является вызовом делегата (§7.6.5.3). Если основное выражение не является ни группой методов, ни значением типа делегата, то возникает ошибка времени привязки.

Необязательный список аргументов (§7.5.1) содержит значения или ссылки на переменные для параметров метода.

Результат вычисления выражения вызова классифицируется следующим образом.

* Если выражение вызова вызывает метод или делегат, который возвращает значение типа void, то результат отсутствует. Выражение без класса допускается только в контексте выражения оператора (§8.6) или в качестве тела лямбда-выражения (§7.15). Иначе возникает ошибка времени привязки.
* Иначе результатом является значение типа, возвращенного методом или делегатом.

#### Вызовы методов

В случае вызова метода основное выражение для выражения вызова должно быть группой методов. Группа методов определяет один вызываемый метод или набор перегруженных методов, из которых будет выбран конкретный вызываемый метод. В последнем случае решение о конкретном вызываемом методе основывается на контексте, предоставляемом типами аргументов в списке аргументов.

Во время привязки обработка вызова метода в виде M(A), где M является группой методов (возможно, включающей список аргументов типов), а A является необязательным списком аргументов, включает следующие этапы.

* Формируется набор методов-кандидатов для вызова метода. Для каждого метода F, связанного с группой методов M:
* Если F является неуниверсальным методом, то F является кандидатом, когда:
* у M нет списка аргументов типа и
* F является применимым в соответствии со списком A (§7.5.3.1).
* Если F является универсальным методом, а у M нет списка аргументов типа, то F является кандидатом, когда:
* успешно завершается вывод типа (§7.5.2), предоставляющий список аргументов типа для вызова, и
* после замены выведенных аргументов типа на соответствующие параметры типа для метода все сформированные типы в списке параметров F соответствуют своим ограничениям (§4.4.4), а список параметров F применим в соответствии со списком A (§7.5.3.1).
* Если F является универсальным методом, а M включает список аргументов типа, то F является кандидатом, когда:
* число параметров типа для метода F равно числу аргументов, предоставленному в списке аргументов типа, и
* после замены аргументов типа на соответствующие параметры типа для метода все сформированные типы в списке параметров F соответствуют своим ограничениям (§4.4.4), а список параметров F применим в соответствии со списком A (§7.5.3.1).
* Набор методов-кандидатов сокращается, и содержи только методы из старших производных типов: для каждого метода C.F в наборе, где C — тип, в котором объявляется метод F, все методы, объявленные в базовом типе C, удаляются из набора. Более того, если C является типом класса, отличным от object, то все методы, объявленные в типе интерфейса из набора удаляются. (Последнее правило применяется только, когда группа методов является результатом поиска члена по параметру типа при наличии действительного базового класса, отличного от object, и непустого действительного набора интерфейсов.)
* Если результирующий набор методов-кандидатов пуст, то дальнейшая обработка вместе со следующими этапами не выполняется, а вместо этого предпринимается попытка обработать вызов в виде вызова метода расширения (§7.6.5.2). Если такая обработка не удается, то применимых методов не существует и возникает ошибка времени привязки.
* Лучший метод из набора методов-кандидатов определяется с помощью правил разрешения перегрузки из раздела §7.5.3. Если определить один лучший метод нельзя, то вызов метода является неоднозначным и возникает ошибка времени привязки. При проведении разрешения перегрузки параметры универсального метода учитываются после замены аргументов типа (предоставленных или выведенных) на соответствующие параметры типа для метода.
* Выполняется последняя проверка выбранного лучшего метода.
* Метод проверяется в контексте группы методов: если наиболее подходящий метод является статическим, группа методов, скорее всего, получилась из простого имени или доступа к члену через тип. Если лучший метод является методом экземпляра, то группа методов должна формироваться на основании простого имени, доступа к члену через переменную или значение или на основании доступа base. Если не выполняется ни одно из этих требований, то возникает ошибка времени привязки.
* Если лучший метод является универсальным методом, то выполняется проверка аргументов типа (предоставленных или выведенных) в соответствии с ограничениями (§4.4.4), объявленными в универсальном методе. Если какой-либо аргумент типа не удовлетворяет соответствующим ограничениям для типа параметра, то возникает ошибка времени привязки.

После выбора и проверки метода в соответствии с указанными этапами на этапе привязки обрабатывается фактический вызов времени выполнения в соответствии с правилами вызова функций-членов из раздела §7.5.4.

Понятным результатом применения правил разрешения, описанных выше, является следующее: Чтобы найти конкретный метод, вызванный при вызове метода, следует начать с типа, указанного в вызове метода, и пройти по цепочке наследования пока не будет найдено объявление последнего применимого, доступного, неперегруженного метода. Затем выполняется вывод типа и разрешение перегрузки для набора применимых, доступных и неперегруженных методов, объявленных в этом типе, и вызывается выбранный таким образом метод. Если метод не найден, предпринимается попытка обработать вызов в виде вызова метода расширения.

#### Вызовы методов расширения

Для вызова метода (§7.5.5.1) одного из видов

expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )

если при обычной обработке вызова применимые методы не найдены, то предпринимается попытка обработать конструкцию в виде вызова метода расширения. Если выражение expr или любой из аргументов args имеет тип времени компиляции dynamic, методы расширения применены не будут.

Задачей является найти лучшее имя типа C, чтобы можно было вызвать соответствующий статический метод:

C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )

Метод расширения Ci.Mj доступен, если:

* Ci не является универсальным или вложенным классом
* Имя Mj является идентификатором.
* Mj доступен и применим для аргументов как статический метод, как указано выше
* Существует неявная идентификация, ссылка или преобразование упаковки от выражения expr до типа первого параметра Mj.

Поиск C выполняется следующим образом.

* Предпринимаются попытки найти набор кандидатов методов расширения, начиная с ближайшего объявления включающего пространства имен, с каждого включающего объявления пространства имен, и заканчивая включающим скомпилированным модулем.
* Если данное пространство имен или блок компиляции напрямую содержит объявления неуниверсального типа Ci с доступными методами расширения Mj, то набор этих методов является набором кандидатов.
* Если пространства имен, импортированные с помощью директив пространства имен в данном пространстве имен или блоке компиляции, напрямую содержат объявления неуниверсального типа Ci с доступными методами расширения Mj, то набор этих методов является набором кандидатов.
* Если ни в одном включающем объявлении пространства имен или скомпилированном модуле набор кандидатов не найден, то возникает ошибка времени компиляции.
* Иначе к набору кандидатов применяется разрешение перегрузки, как описано в разделе (§7.5.3). Если один лучший метод не найден, то возникает ошибка времени компиляции.
* C — это тип, в котором лучший метод объявляется в качестве метода расширения.

После этого с помощью C выполняется вызов метода в виде вызова статического метода (§7.5.4).

Указанные выше правила означают, что методы экземпляра имеют приоритет по сравнению с методами расширения, что методы расширения, доступные во внутренних объявлениях пространств имен, имеют приоритет по сравнению с методами расширения, доступными во внешних объявлениях пространств имен, и что методы расширения, объявленные непосредственно в пространстве имен, имеют приоритет над методами расширения, импортированными в это же пространство имен с помощью директивы пространства имен. Пример:

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)  
 }  
}

В этом примере метод B имеет приоритет по сравнению с первым методом расширения, а метод C имеет приоритет над обоими методами расширения.

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();  
 }  
 }  
}

Вывод в данном примере имеет вид:

E.F(1)  
D.G(2)  
C.H(3)

D.G имеет приоритет над C.G, а E.F имеет приоритет над D.F и C.F.

#### Вызовы делегатов

Для вызова метода основное выражение для выражения вызова должно быть значением типа делегата. Кроме того, учитывая, что тип делегата должен быть функцией-членом с таким же списком параметров, что и тип делегата, он должен быть применим (§7.5.3.1) в соответствии со списком аргументов выражения вызова.

Во время выполнения обработка вызова делегата в виде D(A), где D является основным выражением типа делегата, а A является необязательным списком аргументов, включает следующие этапы.

* Вычисляется D. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Выполняется проверка допустимости значения D. Если D имеет значение null, вызывается исключение System.NullReferenceException и дальнейшие действия не выполняются.
* Иначе D является ссылкой на экземпляр делегата. Вызовы функций-членов (§7.5.4) выполняются для каждой вызываемой сущности в списке вызова делегата. Для вызываемых сущностей, состоящих из экземпляра и метода экземпляра, вызываемый экземпляр является экземпляром, содержащимся в вызываемой сущности.

### Метод доступа к элементу

Доступ к элементу состоит из первичного выражения создания не массива, за которым следует лексема "[" и список аргументов, за которым следует лексема "]". Список аргументов содержит один или несколько аргументов, разделенных запятыми.

element-access:  
primary-no-array-creation-expression [ argument-list ]

Список аргументов метода доступа к элементу не должен содержать аргументы ref или out.

Доступ к элементу динамически привязан (§7.2.2), если верно по меньшей мере одно из следующих утверждений:

* первичное выражение создания не массива имеет тип времени компиляции dynamic;
* по меньшей мере одно выражение списка аргументов имеет тип времени выполнения dynamic, а первичное выражение создания не массива не имеет типа массива.

В этом случае компилятор классифицирует доступ к элементу как значение типа dynamic. Представленные ниже правила определения значения доступа к элементу затем применяются во время выполнения, используя тип времени выполнения вместо правил типа времени компиляции первичного выражения создания не массива и выражений списка аргументов, имеющих тип времени компиляции dynamic. Если основное выражение отличное от создания массива не имеет тип времени компиляции dynamic, то для метода доступа к элементу будет выполнена ограниченная проверка времени компиляции, как описано в §7.5.4.

Если основное выражение отличное от создания массива, для метода доступа к элементу является значением с типом массива, то метод доступа к элементу является методом доступа к массиву (§7.6.6.1). Иначе основное выражение отличное от создания массива, должно быть переменной или значением типа класса, структуры или интерфейса с одним или несколькими членами индексатора, и в таком случае метод доступа к элементу является методом доступа к индексатору (§7.6.6.2).

#### Доступ к массиву

При доступе к массиву основное выражение отличное от создания массива, для метода доступа к элементу должно быть значением типа массива. В дальнейшем список аргументов метода доступа к массиву не должен содержать именованные аргументы. Число выражений в списке аргументов должно быть идентичным рангу типа массива, а каждое выражение должно иметь тип int, uint, long, ulong или должно подразумевать возможность преобразования в один или несколько этих типов.

Результатом вычисления метода доступа к массиву является переменная с типом элемента массива, а именно элемент массива, выбранный по значениям выражений в списке аргументов.

Обработка времени выполнения доступа к массиву в виде P[A], где P является первичным выражением создания не массива с типом массива, а A является списком аргументов, включает следующие этапы.

* Вычисляется P. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Вычисляются выражения индекса в списке аргументов по порядку слева направо. После вычисления всех выражений выполняется неявное преобразование (§6.1) в один из следующих типов:int, uint, long, ulong. Выбирается первый тип из этого списка, для которого существует неявное преобразование. Например, если выражение индекса имеет тип short, выполняется неявное преобразование в тип int, поскольку возможны неявные преобразования из short в int и из short в long. Если при вычислении выражения индекса или последующем неявном преобразовании возникает исключение, то следующие выражения индекса не вычисляются и дальнейшие этапы не выполняются.
* Выполняется проверка допустимости значения P. Если P имеет значение null, вызывается исключение System.NullReferenceException и дальнейшие действия не выполняются.
* Значение каждого выражения в списке аргументов проверяется на соответствие фактическим границам каждого измерения экземпляра массива, на который ссылается P. Если одно или несколько значений выходят за пределы допустимого диапазона, вызывается исключение System.IndexOutOfRangeException и дальнейшие этапы не выполняются.
* Вычисляется расположение элемента массива, заданного выражениями индекса, и оно становится результатом метода доступа к массиву.

#### Доступ к индексатору

При доступе к индексатору массива основное выражение отличное от создания массива для метода доступа к элементу должно быть переменной или значением с типом класса, структуры или интерфейса, и в этом типе должен реализовываться один или несколько индексаторов, применимых в соответствии со списком аргументов для метода доступа к элементу.

Обработка времени привязки доступа к индексатору в виде P[A], где P является первичным выражением создания не массива с типом класса, структуры или интерфейса T, а A является списком аргументов, включает следующие этапы.

* Создается набор индексаторов, предоставляемых T. Этот набор состоит из всех индексаторов, объявленных в T или в базовом для T типе, которые не являются перегруженными объявлениями (override) и доступны в текущем контексте (§3.5).
* Этот набор сокращается до индексаторов, которые применимы и не скрыты другими индексаторами. К каждому индексатору S.I в наборе, где S является типом, в котором определен I, применяются следующие правила.
* Если I не применим в соответствии со списком A (§7.5.3.1), то I удаляется из набора.
* Если I применим в соответствии со списком A (§7.5.3.1), то все индексаторы, объявленные в базовом типе для S удаляются из набора.
* Если I применим в соответствии со списком A (§7.5.3.1) и S имеет тип класса, отличный от object, то из набора удаляются все индексаторы, объявленные в интерфейсе.
* Если результирующий набор индексаторов-кандидатов пуст, то применимых индексаторов нет и возникает ошибка времени привязки.
* Лучший индексатор из набора индексаторов-кандидатов определяется с помощью правил разрешения перегрузки из раздела §7.5.3. Если определить один лучший индексатор нельзя, то метод доступа к индексатору является неоднозначным и возникает ошибка времени привязки.
* Вычисляются выражения индекса в списке аргументов по порядку слева направо. Результатом обработки доступа к индексатору является выражение, классифицированное как доступ к индексатору. Выражение доступа к индексатору ссылается на индексатор, определенный на предыдущем этапе, и имеет связанное выражение экземпляра P и связанный список аргументов A.

В зависимости от контекста использования доступ к индексатору задает вызов метода доступа get или метода доступа set индексатора. Если доступ к индексатору является назначением присваивания, то для назначения нового значения вызывается метод доступа set (§7.17.1). Во всех остальных случаях для получения текущего значения вызывается метод доступа get (§7.1.1).

### Доступ this

Доступ this представляет собой зарезервированное слово this.

this-access:  
this

Доступ this допустим только в блоке конструктора экземпляра, метода экземпляра или метода доступа к экземпляру. Он имеет одно из следующих значений.

* Когда доступ this используется в основном выражении внутри конструктора экземпляра класса, он классифицируется как значение. Типом значения является тип экземпляра (§10.3.1) класса, внутри которого происходит это использование, а значением является ссылка на создаваемый объект.
* Когда доступ this используется в основном выражении внутри метода экземпляра или метода доступа к экземпляру, он классифицируется как значение. Типом значения является тип экземпляра (§10.3.1) класса, внутри которого возникает это использование, а значением является ссылка на объект, для которого вызывается метод или метод доступа.
* Когда доступ this используется в основном выражении внутри конструктора экземпляра структуры, он классифицируется как значение. Типом значения является тип экземпляра (§10.3.1) структуры, внутри которой происходит это использование, а значением является создаваемая структура. Переменная this конструктора экземпляра структуры действует точно так же, как параметр out типа структуры, в частности, это означает, что переменная должна явно назначаться в каждом пути выполнения конструктора экземпляра.
* Когда доступ this используется в основном выражении внутри метода экземпляра или метода доступа к экземпляру структуры, он классифицируется как значение. Типом переменной является тип экземпляра структуры (§10.3.1), внутри которой происходит это использование.
* Если метод или метод доступа не является итератором (§10.14), то переменная this представляет структуру, для которой был вызван метод или метод доступа, и действует точно так же, как параметр ref типа структуры.
* Если метод или метод доступа является итератором, то переменная this представляет копию структуры, для которой был вызван метод или метод доступа, и действует точно так же, как параметр value типа структуры.

Использование слова this в основном выражении в контексте, отличном от указанных выше, приводит к возникновению ошибки времени компиляции. В частности, нельзя ссылаться на this в статическом методе, методе доступа к статическому свойству или в инициализаторе переменной объявления поля.

### Доступ base

Доступ base состоит из зарезервированного слова base, за которым следует либо лексема "." с идентификатором, либо список аргументов, заключенный в квадратные скобки:

base-access:  
base . identifier  
base [ argument-list ]

Доступ base используется для доступа к членам базового класса, скрытым членами с такими же именами в текущем классе или структуре. Доступ base допустим только в блоке конструктора экземпляра, метода экземпляра или метода доступа к экземпляру. Когда вызов base.I происходит в классе или структуре, I должно обозначать член базового класса для данного класса или структуры. Так же, когда вызов base[E] оказывается в классе, в базовом классе должен существовать применимый индексатор.

Во время привязки выражения доступа base в виде base.I и base[E] вычисляются точно так же, как если бы они записывались в виде ((B)this).I и ((B)this)[E], где B является базовым классом для класса или структуры, в которой находится эта конструкция. Таким образом, base.I и base[E] соответствуют выражениям this.I и this[E] за исключением того, что this рассматривается как экземпляр базового класса.

Когда в доступе base оказывается ссылка на виртуальную функцию-член (метод, свойство или индексатор), способ определения вызываемой во время выполнения функции-члена (§7.5.4) меняется. Вызываемая функция-член определяется путем поиска наиболее производной реализации (§10.6.3) функции-члена в соответствии с B (вместо сравнения с типом this во время выполнения, что является обычной процедурой при доступе, отличном от base). Таким образом, внутри виртуальной (virtual) или перегруженной (override) функции-члена доступ base можно использовать для вызова наследованной реализации функции-члена. Если функция-член, на которую ссылается доступ base, является абстрактной, то возникает ошибка времени привязки.

### Постфиксные операторы инкремента и декремента

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

Операнд постфиксного оператора инкремента или декремента должен быть выражением, которое классифицируется как переменная, доступ к свойству или доступ к индексатору. Результатом операции является значение того же типа, что и операнд.

Если первичное выражение имеет тип времени компиляции dynamic, то оператор динамически привязан (§7.2.2), выражение после инкремента или выражение после декремента имеет тип времени компиляции dynamic, а также во время выполнения применяются следующие правила, используя тип времени выполнения первичного выражения.

Если операндом постфиксного оператора инкремента или декремента является свойство или доступ к индексатору, то у свойства и индексатора должны быть оба метода доступа get и set. В противном случае возникает ошибка времени привязки.

Для выбора конкретной реализации оператора используется разрешение перегрузки унарных операторов (§7.3.3). Предопределенные операторы ++ и -- существуют для следующих типов: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal и любых перечисляемых типов. Стандартный оператор ++ возвращает значение, полученное добавлением 1 к операнду, а стандартный оператор -- возвращает значение, полученное вычитанием 1 из операнда. В контексте checked если результат такого сложения или вычитания выходит за пределы допустимого диапазона для типа результата и результат имеет целый тип или тип перечисления, то возникает исключение System.OverflowException.

Во время выполнения обработка постфиксных операций инкремента или декремента в виде x++ или x-- включает следующие этапы.

* Если x классифицируется как переменная, то:
* x вычисляется для создания переменной.
* Значение x сохраняется.
* Вызывается выбранный оператор с сохраненным значением x в качестве аргумента.
* Значение, возвращенное оператором, сохраняется в расположении, предоставленном при вычислении x.
* Сохраненное значение x становится результатом операции.
* Если x классифицируется как свойство или доступ к индексатору, то:
* Вычисляются выражение экземпляра (если x не имеет тип static) и список аргументов (если x является доступом к индексатору), связанные с x, и полученные результаты используются при последующих вызовах методов доступа get и set.
* Вызывается метод доступа get для x, а возвращенное значение сохраняется.
* Вызывается выбранный оператор с сохраненным значением x в качестве аргумента.
* Вызывается метод доступа set для x со значением, возвращенным оператором в качестве своего аргумента value.
* Сохраненное значение x становится результатом операции.

Операторы ++ и -- также могут использоваться в препозиции (§7.7.5). Обычно результатом операторов x++ и x-- является значение x до операции, тогда как результатом ++x и --x является значение x после операции. В обоих случаях сама переменная x имеет одинаковое значение после операции.

Реализацию operator ++ или operator -- можно вызывать в префиксной и постфиксной форме. Для двух этих форм нельзя создать разные реализации операторов.

### Оператор new

Оператор new используется для создания новых экземпляров типов.

Существует три формы выражений new:

* Выражения создания объектов используются для создания новых экземпляров типов класса и типов значения.
* Выражения создания массивов используются для создания новых экземпляров типов массива.
* Выражения создания делегатов используются для создания новых экземпляров типа делегата.

Оператор new подразумевает создание экземпляра типа, но необязательно подразумевает динамическое выделение памяти. В частности, для экземпляров с типом значения не требуется дополнительная память помимо переменных, в которых они находятся, и при использовании new для создания экземпляров с типом значения динамическое выделение памяти не происходит.

#### Выражения создания объектов

Выражение создания объекта используется для создания нового экземпляра типа класса или типа значения.

object-creation-expression:  
new type ( argument-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

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

Тип выражения создания объекта должен быть равен типу класса, типу значения или параметру типа. Тип не может быть типом класса abstract.

Необязательный список аргументов (§7.5.1) допускается, только если тип равен типу класса или типу структуры.

Выражение создания объекта может не содержать списка аргументов конструктора в скобках, если оно включает инициализатор объекта или коллекции. Отсутствие списка аргументов конструктора и скобок эквивалентно заданию пустого списка аргументов.

При обработке выражения создания объекта, которое включает инициализатор объекта или коллекции, сначала выполняется конструктор экземпляра, а затем выполняется инициализация члена или элемента, указанного в инициализаторе объекта (§7.6.10.2) или коллекции (§7.6.10.3).

Если какой-либо из аргументов в необязательном *списке аргументов* имеет тип времени компиляции dynamic, то выражение создания объекта динамически привязано (§7.2.2), и во время выполнения применяются следующие правила, используя тип времени выполнения тех аргументов из списка аргументов, которые имеют тип времени компиляции dynamic. Однако, для создания объекта выполняется ограниченная проверка времени компиляции, как описано в разделе §7.5.4.

Во время привязки обработка выражения создания объекта в виде new T(A), где T является типом класса или типом значения, а A является необязательным списком аргументов, включает следующие этапы.

* Если T является типом значения и список A не указан.
* Выражение создания объекта является вызовом конструктора по умолчанию. Результатом выражения создания объекта является значение типа T, а именно значение T по умолчанию, определенное в §4.1.1.
* Иначе, если T является параметром типа и A не указан.
* Если для T не были указаны ограничения типа значения или конструктора (§10.1.5), то возникает ошибка времени привязки.
* Результатом выражения создания объекта является значение типа времени выполнения, к которому привязан параметр типа, а именно результат вызов конструктора по умолчанию для этого типа. Тип времени выполнения может быть типом значения или ссылочным типом.
* Иначе, если T является типом класса или типом структуры:
* Если T является типом класса с модификатором abstract, возникает ошибка времени компиляции.
* Вызываемый конструктор экземпляра определяется с помощью правил разрешения перегрузки из раздела §7.5.3. Набор кандидатов конструкторов экземпляров включает все доступные конструкторы экземпляров, объявленные в T, применимые в соответствии со списком A (§7.5.3.1). Если набор кандидатов конструкторов экземпляров пуст или невозможно определить один лучший конструктор экземпляра, то возникает ошибка времени привязки.
* Результатом выражения создания объекта является значение типа T, а именно значение, получаемое при вызове конструктора экземпляра, определенного на предыдущем этапе.
* Иначе выражение создания объекта является недопустимым и возникает ошибка времени привязки.

Даже если выражение создания объекта динамически привязано, тип времени компиляции все равно будет T.

Во время выполнения обработка выражения создания объекта в виде new T(A), где T является типом класса или типом структуры, а A является необязательным списком аргументов, включает следующие этапы.

* Если T является типом класса:
* Создается новый экземпляр класса T. Если для создания нового экземпляра недостаточно памяти, то возникает исключение System.OutOfMemoryException и дальнейшие этапы не выполняются
* Все поля нового экземпляра инициализируются с помощью значений по умолчанию (§5.2).
* В соответствии с правилами вызова функции-члена (§7.5.4) вызывается конструктор экземпляра. Ссылка на созданный экземпляр автоматически передается конструктору экземпляра, и к этому экземпляру можно обращаться из этого конструктора с помощью this.
* Если T является типом структуры:
* С помощью выделения временной локальной переменной создается экземпляр типа T. Поскольку для явного присвоения значений каждому полю создаваемого экземпляра требуется конструктор экземпляра типа структуры, инициализация временной переменной не требуется.
* В соответствии с правилами вызова функции-члена (§7.5.4) вызывается конструктор экземпляра. Ссылка на созданный экземпляр автоматически передается конструктору экземпляра, и к этому экземпляру можно обращаться из этого конструктора с помощью this.

#### Инициализаторы объектов

Инициализатор объекта задает значения для нуля или нескольких полей или свойств объекта.

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

Инициализатор объекта состоит из последовательности инициализаторов членов, заключенных между лексемами { и } и разделенных запятыми. В каждом инициализаторе члена должно указываться имя доступного поля или свойства инициализируемого объекта, за которым следует знак равенства и выражение или инициализатор объекта или коллекции. Инициализатор объекта не может содержать более одного инициализатора члена для одного поля или свойства. Инициализатор объекта не может ссылаться на инициализируемый им объект.

Инициализатор члена, в котором после знака равенства указывается выражение, обрабатывается так же, как присваиванием (§7.17.1) полю или свойству.

Инициализатор члена, в котором после знака равенства указывается инициализатор объекта, является инициализатором вложенного объекта, то есть выполняет инициализацию внедренного объекта. Вместо присваивания нового значения полю или свойству присваивания в инициализаторе вложенного объекта рассматриваются как присваивания членам поля или свойства. Инициализаторы вложенного объекта не могут применяться к свойствам типа значения или к полям с типом значения, доступным только для чтения.

Инициализатор члена, в котором после знака равенства указывается инициализатор коллекции, выполняет инициализацию внедренной коллекции. Вместо назначения полю или свойству новой коллекции элементы, указанные в инициализаторе, добавляются в коллекцию, на которую ссылается поле или свойство. Поле или свойство должно иметь тип коллекции, который удовлетворяет требованиям раздела §7.6.10.3.

Следующий класс представляет собой точку с двумя координатами.

public class Point  
{  
 int x, y;

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

Экземпляр Point можно создать и инициализировать следующим образом:

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

что равносильно

Point \_\_a = new Point();  
\_\_a.X = 0;  
\_\_a.Y = 1;   
Point a = \_\_a;

где \_\_a является невидимой и недоступной другим образом временной переменной. Следующий класс представляет собой прямоугольник, создаваемый на основании двух точек:

public class Rectangle  
{  
 Point p1, p2;

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

Экземпляр Rectangle можно создать и инициализировать следующим образом:

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

что равносильно

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;

где \_\_r, \_\_p1 и \_\_p2 являются временными переменными, не видимыми и недоступными другим образом.

Если конструктор Rectangle’s выделяет два внедренных экземпляра Point

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

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

то вместо назначения новых экземпляров для инициализации внедренных экземпляров Point можно использовать следующую конструкцию:

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

что равносильно

Rectangle \_\_r = new Rectangle();  
\_\_r.P1.X = 0;  
\_\_r.P1.Y = 1;  
\_\_r.P2.X = 2;  
\_\_r.P2.Y = 3;  
Rectangle r = \_\_r;

#### Инициализаторы коллекции

Инициализатор коллекции задает элементы коллекции.

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

Инициализатор коллекции состоит из последовательности инициализаторов элементов, заключенных между лексемами { и } и разделенных запятыми. Каждый инициализатор элемента задает элемент, добавляемый в инициализируемый объект коллекции, и состоит из списка выражений, заключенных между лексемами { и } и разделенных запятыми. Инициализатор элемента с одним выражением можно записывать без скобок, но в таком случае оно не может быть выражением присваивания, чтобы избежать неоднозначности с инициализаторами членов. Создание выражения не присваивания определяется в разделе §7.18.

Ниже приводится пример выражения создания объекта, в которое входит инициализатор коллекции:

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

Объект коллекции, к которому применяется инициализатор коллекции, должен иметь тип, в котором реализуется интерфейс System.Collections.IEnumerable, иначе будет возникать ошибка времени компиляции. Для каждого указанного элемента по порядку инициализатор вызывает метод Add целевого объекта со списком выражений инициализатора элемента в качестве списка аргументов, применяя обычное разрешение перегрузки для каждого вызова. Таким образом, объект коллекции должен содержать применимый метод Add для каждого инициализатора элемента.

Следующий класс представляет собой контакт с именем и списком телефонных номеров:

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; } }  
}

Экземпляр List<Contact> можно создать и инициализировать следующим образом:

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" }  
 }  
};

что равносильно

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;

где \_\_clist, \_\_c1 и \_\_c2 являются временными переменными, не видимыми и недоступными другим образом.

#### Выражения создания массива

Выражение создания массива используется для создания нового экземпляра типа массива.

array-creation-expression:  
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

Выражение создания массива первого типа создает экземпляр массива с типом, который получается после удаления всех отдельных выражений из списка выражений. Например, выражение создания массива new int[10, 20] создает экземпляр массива с типом int[,], а выражение new int[10][,] — экземпляр массива с типом int[][,]. Каждое выражение в списке выражений должно иметь тип int, uint, long, ulong или тип, который может быть явно преобразован в один или несколько этих типов. Значение каждого выражения определяет размер соответствующего измерения в новом созданном экземпляре массива. Поскольку размер измерения массива должен быть неотрицательным, при наличии константного выражения с отрицательным значением в списке выражений будет возникать ошибка времени компиляции.

За исключением небезопасных контекстов (§18.1) формат массива не указывается.

Если выражение создания массива первого типа содержит инициализатор массива, то каждое выражение в списке выражений должно быть константным, а ранг и длина измерения, указанные в списке выражений, должны совпадать с соответствующими значениями инициализатора массива.

В выражении создания массива второго или третьего типа ранг указанного типа массива или спецификация ранга должны быть равны соответствующим значениям инициализатора массива. Длины отдельных измерений выводятся из числа элементов в каждом соответствующем вложенном уровне инициализатора массива. Таким образом, выражение

new int[,] {{0, 1}, {2, 3}, {4, 5}}

в точности соответствует

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}

Выражение создания массива третьего типа называется выражением создания массива с неявным указанием типа. Оно похоже на второй тип за исключением того, что тип элемента массива не задается явно, но определяется как наиболее общий тип (§7.5.2.14) в наборе выражений в инициализаторе массива. Для многомерного массива (в спецификации ранга, у которого есть, по крайней мере, одна запятая) этот набор включает все выражения, находящиеся во вложенных инициализаторах массива.

Инициализаторы массива подробнее описываются далее в разделе §12.6.

Результат вычисления выражения создания массива классифицируется как значение, а именно как ссылка на новый созданный экземпляр массива. Во время выполнения обработка выражения создания массива включает следующие этапы.

* Вычисляются выражения длины измерений в списке выражений по порядку слева направо. После вычисления всех выражений выполняется неявное преобразование (§6.1) в один из следующих типов:int, uint, long, ulong. Выбирается первый тип из этого списка, для которого существует неявное преобразование. Если при вычислении выражения или последующем неявном преобразовании возникает исключение, то следующие выражения не вычисляются и дальнейшие этапы не выполняются.
* Вычисленные значения для длин измерений проверяются следующим образом. Если одно или несколько значений оказываются меньше нуля, то вызывается исключение System.OverflowException и дальнейшие этапы не выполняются.
* Создается экземпляр массива с полученными длинами измерений. Если для создания нового экземпляра недостаточно памяти, то возникает исключение System.OutOfMemoryException и дальнейшие этапы не выполняются
* Все элементы нового экземпляра массива инициализируются с помощью значений по умолчанию (§5.2).
* Если выражение создания массива содержит инициализатор массива, то вычисляется каждое выражение в инициализаторе массива и полученное значение назначается соответствующему элементу массива. Вычисления и присваивания выполняются в порядке записи выражений в инициализаторе массива, другими словами, инициализация элементов происходит по возрастанию индекса, причем первым обрабатывается самое правое измерение. Если при вычислении данного выражения или последующем присваивании соответствующему элементу массива возникает исключение, то другие элементы не инициализируются (следовательно, оставшиеся элементы будут иметь значения по умолчанию).

Выражение создания массива позволяет проводить инициализацию массива с помощью элементов типа массива, однако элементы такого массива необходимо инициализировать вручную. Например, в выражении

int[][] a = new int[100][];

создается одномерный массив со 100 элементами типа int[]. Исходным значением каждого элемента является null. В этом же выражении создания массива нельзя инициализировать подмассивы, и выражение

int[][] a = new int[100][5]; // Error

возникает ошибка времени компиляции. Вместо этого инициализация подмассивов должна выполняться вручную, ср.:

int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];

Когда массив массивов имеет «прямоугольную» форму, то есть когда все подмассивы имеют одинаковую длину, более эффективно использовать многомерный массив. В примере выше при инициализации массива массивов создается 101 объект — один внешний массив и 100 вложенных массивов. Напротив, в выражении

int[,] = new int[100, 5];

создается только один объект, двумерный массив, и это создание выполняется в одном выражении.

Ниже приведены примеры выражений создания массивов с неявным заданием типа.

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

Последнее выражение вызывает ошибку времени выполнения, поскольку типы int и string не могут быть неявно преобразованы один в другой, поэтому в данном случае наиболее общий тип отсутствует. В этом случае необходимо использовать выражение создания массива с явным заданием типа, например, указав тип object[]. Иначе один из элементов можно привести к общему базовому типу, который затем станет выведенным типом элемента.

Выражения создания массива с неявным заданием типа можно комбинировать с инициализаторами анонимных объектов (§7.6.10.6) для создания структуры данных с анонимным типом Пример:

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

#### Выражения создания делегата

Выражение создания делегата используется для создания нового экземпляра типа делегата.

delegate-creation-expression:  
new delegate-type ( expression )

Аргументом выражения создания делегата должна быть группа методов, анонимная функция, или значение типа времени компиляции dynamic или типа делегата. Если аргумент является группой методов, он определяет метод, а для метода экземпляра — объект, для которого создается делегат. Если аргументом является анонимная функция, он напрямую определяет параметры и тело метода целевого делегата. Если аргумент является значением, он определяет экземпляр делегата, для которого создается копия.

Если выражение имеет тип dynamic во время компиляции, то выражение создания делегата динамически связано (§7.2.2), а правила внизу применяются во время выполнения с помощью типа времени выполнения выражения. Иначе правила применяются во время компиляции.

Во время привязки обработка выражения создания делегата в виде new D(E), где D имеет тип делегата, а E является выражением, включает следующие этапы.

* Если E является группой методов, выражение создания делегата обрабатывается так же, как и преобразование группы методов (§6.6) из E в D.
* Если E является анонимной функцией, выражение создания делегата обрабатывается так же, как и преобразование анонимной функции (§6.5) из E в D.
* Если E является значением, то E должно быть совместимо (§15.1) с D, а результатом является новый созданный делегат типа D, который ссылается на тот же список вызова, что и E. Если E не совместимо с D, возникает ошибка времени компиляции.

Во время выполнения обработка выражения создания делегата в виде new D(E), где D имеет тип делегата, а E является выражением, включает следующие этапы.

* Если E является группой методов, выражение создания делегата обрабатывается как преобразование группы методов (§6.6) из E в D.
* Если E является анонимной функцией, выражение создания делегата обрабатывается как преобразование анонимной функции из E в D (§6.5).
* Если E является значением типа делегата:
* Вычисляется E. Если при этом вычислении возникает исключение, дальнейшие этапы не выполняются.
* Если E имеет значение null, вызывается исключение System.NullReferenceException и дальнейшие действия не выполняются.
* Создается новый экземпляр типа делегата D. Если для создания нового экземпляра недостаточно памяти, то возникает исключение System.OutOfMemoryException и дальнейшие этапы не выполняются
* Новый экземпляр делегата инициализируется с помощью того же списка вызова, что и экземпляр делегата, предоставленный E.

Список вызова делегата определяется при инициализации делегата и остается неизменным в течение всего срока жизни делегата. Другими словами после создания делегата изменить его целевые вызываемые сущности невозможно. При объединении двух делегатов или удалении одного из другого (§15.1) создается новый делегат. Содержимое существующих делегатов не меняется.

Нельзя создать делегат, который бы ссылался на свойство, индексатор, пользовательский оператор, конструктор экземпляра, деструктор или статический конструктор.

Как говорилось выше, когда делегат создается из группы методов, список формальных параметров и тип возвращаемого значения делегата определяют выбираемый перегруженный метод. В этом примере

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;  
 }  
}

поле A.f инициализируется с помощью делегата, который ссылается на второй метод Square, потому что этот метод в точности совпадает по списку формальных параметров и типу возвращаемого значения с DoubleFunc. Если бы второй метод Square отсутствовал, то возникла бы ошибка времени компиляции.

#### Выражения создания анонимных объектов

Выражение создания анонимного объекта используется для создания объекта анонимного типа.

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

Инициализатор анонимного объекта объявляет анонимный тип и возвращает экземпляр этого типа. Анонимный тип — это безымянный тип класса, который наследуется непосредственно от класса object. Члены анонимного типа представляют собой последовательность свойств, доступных только на чтение, выводимых из инициализатора анонимного объекта, использованного для создания экземпляра типа. В частности, инициализатор анонимного объекта вида

new { p1 = e1 , p2 = e2 , … pn = en }

объявляет анонимный тип вида

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() { … }  
}

где каждый Tx является типом соответствующего выражения ex. Выражение, используемое в деклараторе члена, должно иметь тип. Поэтому, если выражение в деклараторе члена равно NULL или является анонимной функцией, возникает ошибка времени компиляции. Ошибка времени компиляции также возникает, если выражение имеет небезопасный тип.

Имена анонимного типа и параметра его метода Equals автоматически создаются компилятором, и на них нельзя ссылаться в тексте программы.

В рамках одной программы два инициализатора, которые задают последовательность свойств с одинаковыми именами и типами времени компиляции в одном порядке, создадут экземпляры одного анонимного типа.

В этом примере

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

присваивание на последней строке допускается, потому что p1 и p2 имеют один анонимный тип.

Методы Equals и GetHashcode для анонимных типов перегружают методы, унаследованные от класса object, и определяются через методы Equals и GetHashcode для свойств таким образом, что два экземпляра одного анонимного типа будут равны, только если равны все их свойства.

Название декларатора члена можно сократить до простого имени (§7.5.2), доступа к члену (§7.5.4) или доступа base (§7.6.8). Такой способ называется инициализацией проекции и представляет собой удобный способ для объявления присваивания свойству с таким же именем. В частности, деклараторы членов вида

identifier expr . identifier

в точности совпадают со следующими выражениями соответственно:

identifer = identifier identifier = expr . identifier

Таким образом, при инициализации проекции идентификатор позволяет выбирать и значение, и поле или свойство, которому присвоено это значение. С интуитивной точки зрения при инициализации проекции проецируется не только значение, но и имя значения.

### Оператор typeof

Оператор typeof используется для получения объекта System.Type для типа.

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 ,

Выражение typeof первого типа состоит из ключевого слова typeof, за которым следует тип в скобках. Результатом выражения этого типа является объект System.Type для указанного типа. Для любого типа существует только один объект System.Type. Это означает, что для типа T всегда выполняет равенство typeof(T) == typeof(T). Тип не может быть dynamic.

Выражение typeof второго типа состоит из ключевого слова typeof, за которым следует непривязанное имя типа в скобках. Непривязанное имя типа очень похоже на имя типа (§3.8) за исключением того, что непривязанное имя типа содержит универсальные спецификаторы измерений, тогда как имя типа содержит списки аргументов типа. Если операнд выражения typeof является последовательностью токенов, соответствующей правилам формирования непривязанного имени типа и имени типа, а именно если он не содержит ни универсальный спецификатор измерения, ни список аргументов типа, то последовательность токенов считается именем типа. Значение непривязанного имени типа определяется следующим образом.

* Последовательность лексем преобразуется в имя типа путем замены каждого универсального спецификатора измерения на список аргументов типа с таким же числом запятых и ключевым словом object, как и у каждого аргумента типа.
* Получившееся имя типа вычисляется, причем все ограничения параметра типа игнорируются.
* Непривязанное имя типа разрешается в непривязанный универсальный метод, связанный с результирующим сформированным типом (§4.4.3).

Результатом выражения typeof является объект System.Type для результирующего непривязанного универсального типа.

Выражение typeof третьего типа состоит из ключевого слова typeof, за которым следует ключевое слово void в скобках. Результатом выражения этого типа является объект System.Type, обозначающий отсутствие типа. Объект типа, возвращаемый выражением typeof(void), отличается от объекта типа, возвращаемого для любого типа. Этот специальный объект типа удобно использовать в библиотеках классов, которые допускают отражение в методы языка, где для этих методов желательно иметь способ представления типа возвращаемого значения, включая методы, возвращающие void, с помощью экземпляра System.Type.

Оператор typeof можно использовать в параметре типа. Результатом является объект System.Type для типа времени выполнения, который был связан с параметром типа. Оператор typeof также можно использовать в сформированном типе или непривязанном универсальном типе (§4.4.3). Объект System.Type для непривязанного универсального типа отличается от объекта System.Type для типа экземпляра. Тип экземпляра всегда является закрытым сформированным типом во время выполнения, поэтому его объект System.Type зависит от используемых во время выполнения аргументов типа, тогда как у непривязанного универсального типа аргументов типа нет.

Пример:

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();  
 }  
}

получается следующий вывод:

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]

Обратите внимание, что int и System.Int32 являются одним типом.

Также обратите внимание, что результат выражения typeof(X<>) не зависит от аргумента типа, а результат typeof(X<T>) зависит.

### Операторы checked и unchecked

Операторы checked и unchecked используются для управления контекстом контроля переполнения в арифметических операциях и преобразованиях целых типов.

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

Оператор checked вычисляет содержащееся в нем выражение в проверяемом контексте, а оператор unchecked — в непроверяемом. Выражение checked или выражение unchecked в точности соответствует выражению в скобках (§7.6.3) за исключением того, что содержащееся выражение вычисляется в указанном контексте проверки переполнения.

Контекстом проверки переполнения также можно управлять с помощью операторов checked и unchecked (§8.11).

Контекст проверки переполнения, заданный с помощью операторов и инструкций checked и unchecked, затрагивает следующие операции.

* Стандартные унарные операторы ++ и -- (§7.6.9 и §7.7.5), когда операнд имеет целый тип.
* Стандартный унарный оператор - (§7.7.2), когда операнд имеет целый тип.
* Стандартные бинарные операторы +, -, \* и / (§7.8), когда оба операнда имеют целый тип.
* Явное числовое преобразование (§6.2.1) из одного целого типа в другой или из типа float или double в целый тип.

Когда в результате одной из указанных операций получается слишком большой результат для его отражения в целевом типе, получившимся поведением управляет контекст выполнения операции.

* В контексте checked, если операция является константным выражением (§7.19), возникает ошибка времени компиляции. Иначе, когда операция выполняется во время выполнения, возникает исключение System.OverflowException.
* В контексте unchecked результат усекается путем отбрасывания всех старших битов, которые не помещаются в целевой тип.

Для неконстантных выражений (выражений, которые вычисляются во время выполнения), не заключенных в операторы или инструкции checked или unchecked, контекстом проверки переполнения по умолчанию является unchecked, если внешние факторы (например, переключатели компилятора и конфигурация среды исполнения) не заставляют установить контекст checked.

Для константных выражений (выражений, которые можно полностью вычислить во время компиляции) контекстом проверки переполнения по умолчанию всегда является checked. Если только константное выражение не будет явно помещено в контекст unchecked, переполнения, возникающие при вычислении выражения во время компиляции переполнения, всегда будут вызывать ошибки времени компиляции.

Тело анонимной функции не затрагивается контекстами checked или unchecked, в которых находится анонимная функция.

В этом примере

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  
 }  
}

ошибка времени компиляции не возникает, поскольку ни одно из выражений нельзя вычислить во время компиляции. Во время выполнения метод F вызывает исключение System.OverflowException, а метод G возвращает значение –727379968 (младшие 32 бита результата, выходящего за пределы диапазона). Поведение метода H зависит от контекста проверки переполнения, заданного по умолчанию для компиляции, но оно будет совпадать либо с F, либо с G.

В этом примере

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  
 }  
}

переполнения, возникающие при вычислении константных выражений в F и H, вызывают возникновение ошибок времени компиляции, потому что выражения вычисляются в контексте checked. При вычислении константного выражения в G также может произойти переполнение, но поскольку вычисление выполняется в контексте unchecked, о переполнении не сообщается.

Операторы checked и unchecked влияют на контекст проверки переполнения только для тех операций, которые в текстовом виде заключены между лексемами «(« и «)». Эти операторы не влияют на функции-члены, вызываемые в результате вычисления содержащегося выражения. В этом примере

class Test  
{  
 static int Multiply(int x, int y) {  
 return x \* y;  
 }

static int F() {  
 return checked(Multiply(1000000, 1000000));  
 }  
}

использование checked в F не влияет на вычисление выражения x \* y в Multiply, поэтому выражение x \* y вычисляется в контексте проверки переполнения по умолчанию.

Оператор unchecked удобно использовать при создании написания констант подписанных целых типов в шестнадцатеричной записи. Пример:

class Test  
{  
 public const int AllBits = unchecked((int)0xFFFFFFFF);

public const int HighBit = unchecked((int)0x80000000);  
}

Обе шестнадцатеричные константы выше имеют тип uint. Поскольку константы выходят за пределы диапазона int, без оператора unchecked попытки приведения типа к int будут вызывать ошибки времени компиляции.

Операторы и инструкции checked и unchecked позволяют разработчику управлять определенными аспектами некоторых числовых вычислений. Однако поведение некоторых числовых операторов зависит от типов данных их операндов. Например, умножение двух десятичных чисел всегда приводит к возникновению исключения при переполнении даже внутри явно указанной конструкции unchecked. А умножение двух чисел с плавающей запятой никогда не приводит к возникновению исключения при переполнении даже внутри явно указанной конструкции checked. Кроме того, другие операторы никогда не затрагиваются режимом проверки, заданным явно или по умолчанию.

### Выражения значения по умолчанию

Выражение значения по умолчанию используется для получения значения по умолчанию (§5.2) для типа. Обычно выражение значения по умолчанию используется для параметров типа, поскольку может быть неизвестно, имеет ли параметр типа тип значения или ссылочный тип. (Если тип параметра типа отличается от ссылочного, то преобразования из литерала null в параметр типа не существует.)

default-value-expression:  
default ( type )

Если тип в выражении значения по умолчанию во время выполнения оказывается ссылочным типом, то результатом выражения является значение null, преобразованное в этот тип. Если тип в выражении значения по умолчанию во время выполнения оказывается типом значения, то результатом выражения является значение типа значения по умолчанию (§4.1.2).

Выражение значения по умолчанию является константным выражением (§7.19), если тип является ссылочным или параметром типа, который является ссылочным типом (§10.1.5). Кроме того, выражение значения по умолчанию является константным выражением, если его тип является одним из следующих типов значений: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool или любым перечисляемым типом.

### Выражения анонимного метода

Выражение анонимного метода представляет собой один из двух способов определения анонимной функции. Они описываются дальше в разделе §7.15.

## Унарные операторы.

Операторы +, -, !, ~, ++, --, операторы приведения типа и операторы await называются унарными операторами.

unary-expression:  
primary-expression  
+ unary-expression  
- unary-expression  
! unary-expression  
~ unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression  
await-expression

Если операнд унарного выражения имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции унарного выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения операнда.

### Унарный оператор «плюс»

Для операции вида +x чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки унарного оператора (§7.3.3). Операнд преобразуется в тип параметра выбранного оператора, а тип результата является типом возвращаемого значения этого оператора. К cтандартным унарным операторам «плюс» относятся:

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);

Для каждого из этих операторов результатом является просто значение операнда.

### Унарный оператор «минус»

Для операции вида –x чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки унарного оператора (§7.3.3). Операнд преобразуется в тип параметра выбранного оператора, а тип результата является типом возвращаемого значения этого оператора. К cтандартным операторам взятия обратного знака относятся:

* Взятие обратного знака для целых чисел:

int operator –(int x);  
long operator –(long x);

Результат вычисляется вычитанием x из нуля. Если значение x является наименьшим представимым значением для типа операнда (−231 для int или −263 для long)), то математическая операция взятия обратного знака для x не представима в типе операнда. Если это происходит в контексте checked, то возникает исключение System.OverflowException. Если это происходит в контексте unchecked, то результатом является значение операнда и о переполнении не сообщается.

Если операнд оператора взятия обратного знака имеет тип uint, то он преобразуется в тип long и результат имеет тип long. Исключением является правило, которое позволяет записывать значение типа int −2147483648 (−231) в виде литерала десятичного целого числа (§2.4.4.2).

Если операнд оператора взятия обратного знака имеет тип ulong, возникает ошибка времени компиляции. Исключением является правило, которое позволяет записывать значение типа long −9223372036854775808 (−263) в виде литерала десятичного целого числа (§2.4.4.2).

* Взятие обратного знака для чисел с плавающей запятой:

float operator –(float x);  
double operator –(double x);

Результатом является значение x с обратным знаком. Если x является NaN, то результат также равен NaN.

* Взятие обратного знака для десятичных чисел:

decimal operator –(decimal x);

Результат вычисляется вычитанием x из нуля. Взятие обратного знака для десятичных чисел эквивалентно использованию унарного оператора «минус» типа System.Decimal.

### Оператор логического отрицания

Для операции вида !x чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки унарного оператора (§7.3.3). Операнд преобразуется в тип параметра выбранного оператора, а тип результата является типом возвращаемого значения этого оператора. Существует единственный стандартный оператор логического отрицания:

bool operator !(bool x);

Этот оператор вычисляет логическое отрицание операнда. Если операнд равен true, то результат равен false. Если значение операнда равно false, результат равен true.

### Оператор побитового дополнения

Для операции вида ~x чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки унарного оператора (§7.3.3). Операнд преобразуется в тип параметра выбранного оператора, а тип результата является типом возвращаемого значения этого оператора. К стандартным операторам побитового дополнения относятся:

int operator ~(int x);  
uint operator ~(uint x);  
long operator ~(long x);  
ulong operator ~(ulong x);

Для каждого из этих операторов результатов операции является побитовое дополнение x.

Каждый тип перечисления E неявно предоставляет следующий оператор побитового дополнения:

E operator ~(E x);

Результат вычисления x, где x является выражением перечисляемого типа E с базовым типом U, в точности равен значению (E)(~(U)x), за исключением того, что преобразование в E всегда выполняется как в контексте unchecked (§7.6.12).

### Префиксные операторы инкремента и декремента

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

Операнд префиксного оператора инкремента или декремента должен быть выражением, которое классифицируется как переменная, доступ к свойству или доступ к индексатору. Результатом операции является значение того же типа, что и операнд.

Если операндом префиксного оператора инкремента или декремента является свойство или доступ к индексатору, то у свойства и индексатора должны быть оба метода доступа get и set. В противном случае возникает ошибка времени привязки.

Для выбора конкретной реализации оператора используется разрешение перегрузки унарных операторов (§7.3.3). Предопределенные операторы ++ и -- существуют для следующих типов: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal и любых перечисляемых типов. Стандартный оператор ++ возвращает значение, полученное добавлением 1 к операнду, а стандартный оператор -- возвращает значение, полученное вычитанием 1 из операнда. В контексте checked если результат такого сложения или вычитания выходит за пределы допустимого диапазона для типа результата и результат имеет целый тип или тип перечисления, то возникает исключение System.OverflowException.

Во время выполнения обработка префиксных операций инкремента или декремента в виде ++x или --x включает следующие этапы.

* Если x классифицируется как переменная, то:
* x вычисляется для создания переменной.
* Вызывается выбранный оператор со значением x в качестве аргумента.
* Значение, возвращенное оператором, сохраняется в расположении, предоставленном при вычислении x.
* Значение, возвращенное оператором, становится результатом операции.
* Если x классифицируется как свойство или доступ к индексатору, то:
* Вычисляются выражение экземпляра (если x не имеет тип static) и список аргументов (если x является доступом к индексатору), связанные с x, и полученные результаты используются при последующих вызовах методов доступа get и set.
* Вызывается метод доступа get для x.
* Вызывается выбранный оператор со значением аргумента, равным возвращенному значению метода доступа get.
* Вызывается метод доступа set для x со значением, возвращенным оператором в качестве своего аргумента value.
* Значение, возвращенное оператором, становится результатом операции.

Операторы ++ и --- также могут использоваться в постпозиции (§7.6.9). Обычно результатом операторов x++ и x-- является значение x до операции, тогда как результатом ++x и --x является значение x после операции. В обоих случаях сама переменная x имеет одинаковое значение после операции.

Реализацию operator ++ или operator -- можно вызывать в префиксной и постфиксной форме. Для двух этих форм нельзя создать разные реализации операторов.

### Выражения приведения типа

Выражение приведения типа используется для явного преобразования выражения в данный тип.

cast-expression:  
( type ) unary-expression

Выражение приведения типа вида (T)E, где T является типом, а E — унарным выражением, выполняет явное преобразование (§6.2) значения E в тип T. Если явное преобразование из E в T отсутствует, то возникает ошибка времени привязки. Иначе результатом является значение, полученное при явном преобразовании. Результат всегда классифицируется как значение, даже если E обозначает переменную.

Запись выражения приведения типа создает некоторую синтаксическую неоднозначность. Например, выражение (x)–y можно интерпретировать как выражение приведения типа (приведение –y к типу x) или как аддитивное выражение с выражением в скобках (в котором вычисляется значение x – y).

Для разрешения неоднозначности в выражении приведения типа существует следующее правило. Последовательность из одной или нескольких лексем (§2.3.3), заключенная в круглые скобки, считается началом выражения приведения типа, только если верно одно из следующих условий:

* Последовательность лексем имеет правильную грамматику для типа, но не для выражения.
* Последовательность лексем имеет правильную грамматику для типа, и лексема, непосредственно следующая за закрывающей скобкой, равна «~», «!», «(», идентификатору (§2.4.1), литералу (§2.4.4) или любому ключевому слову (§2.4.3) за исключением as и is.

Термин «правильная грамматика» выше означает только то, что последовательность лексем должна соответствовать конкретному грамматическому выводу. В нем специально не учитывается фактическое значение каких-либо составляющих идентификаторов. Например, если x и y являются идентификаторами, то выражение x.y имеет правильную грамматику для типа, даже если x.y фактически не означает тип.

Из правила устранения неоднозначности следует, что если x и y являются идентификаторами, то (x)y, (x)(y) и (x)(-y) являются выражениями приведения типа, а (x)-y не является таким выражением, даже если x обозначает тип. Однако если x является ключевым словом, которое обозначает стандартный тип (например, int), то все четыре вида выражения являются выражениями приведения типа (потому что такое ключевое слово не может само быть выражением).

### Выражения await

Оператор await используется для приостановки вычисления содержащей его асинхронной функции до завершения асинхронной операции, представленной операндом.

await-expression:  
await unary-expression

Выражения await можно использовать только в теле асинхронной функции (§10.14). В рамках ближайшей содержащей его асинхронной функции выражение await не может находится в следующих местах:

* внутри вложенной (не являющейся асинхронной) анонимной функции;
* в блоке catch или finally оператора try;
* внутри блока операторов lock;
* в небезопасном контексте.

Обратите внимание, что выражение await не может находится в большинстве мест внутри выражения запроса, поскольку эти выражения синтаксически преобразуются для использования лямбда-выражений, не являющихся асинхронными.

Внутри асинхронной функции await нельзя использовать как идентификатор. Это позволяет устранить все синтаксические неоднозначности между выражениями await и различными выражениями, использующими идентификаторы. За пределами асинхронных функций await работает как обычный идентификатор.

Операнд выражения await называется задачей. Он представляет асинхронную операцию, которая может быть завершена или не завершена во время вычисления выражения await. Оператор await предназначен для приостановки выполнения содержащей его асинхронной функции до завершения ожидаемой задачи и получения ее результата.

#### Выражения с ожиданием

Задача выражения await должна быть ожидаемым объектом. Выражение *t* является ожидаемым, если выполняется одно из следующих условий.

* *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* реализует интерфейс System.Runtime.CompilerServices.INotifyCompletion (в дальнейшем для краткости называемый INotifyCompletion).
* *A* имеет доступное для чтения свойство экземпляра IsCompleted типа bool.
* *A* имеет доступный метод экземпляра GetResult без параметров и параметров типа.

Метод GetAwaiter предназначен для получения ожидающего объекта для задачи. Тип *A* называется типом ожидающего объекта для выражения await.

С помощью свойства IsCompleted можно определить, завершена ли задача. Если она завершена, нет смысла приостанавливать вычисление.

Метод INotifyCompletion.OnCompleted необходим для того, чтобы подписаться на "продолжение" задачи, то есть на делегат (типа System.Action), который будет вызван после завершения задачи.

С помощью метода GetResult можно получить результат задачи после ее завершения. Результатом может быть успешное завершение, иногда с результирующим значением, или исключение, создаваемое методом GetResult.

#### Классификация выражений await

Классификация выражений await *t* не отличается от классификации выражений (*t*).GetAwaiter().GetResult(). Например, если метод GetResult имеет тип возвращаемого значения void, выражение await классифицируется как Nothing. Если метод имеет тип возвращаемого значения *T*, отличный от void, выражение await классифицируется как значение типа *T*.

#### Вычисление выражений await во время выполнения

Во время выполнения выражение await *t* вычисляется следующим образом.

* Ожидающий объект *a* получается путем вычисления выражения (*t*).GetAwaiter().
* Значение *b* типа bool получается путем вычисления выражения (*a*).IsCompleted.
* Если значение *b* равно false, вычисление зависит от того, реализует ли объект *a* интерфейс System.Runtime.CompilerServices.ICriticalNotifyCompletion (далее для краткости называемый ICriticalNotifyCompletion). Эта проверка выполняется во время привязки, то есть во время выполнения, если объект *a* имеет тип времени компиляции dynamic, или во время компиляции во всех других случаях. Назовем *r* делегатом возобновления (§10.14):
* Если *a* не реализует интерфейс ICriticalNotifyCompletion, вычисляется выражение   
  (*a* as (INotifyCompletion)).OnCompleted(*r*).
* Если *a* реализует интерфейс ICriticalNotifyCompletion, вычисляется выражение   
  (*a* as (ICriticalNotifyCompletion)).UnsafeOnCompleted(*r*).
* Затем вычисление приостанавливается, и управление возвращается текущему вызывающему объекту асинхронной функции.
* Сразу вслед за этим (если значение *b* было равно true) или позднее, после вызова делегата возобновления (если значение *b* было равно false), вычисляется выражение (*a*).GetResult(). Если это выражение возвращает значение, данное значение является результатом *выражения await*. В противном случае результат равен Nothing.

Ожидающий объект должен реализовывать методы интерфейса INotifyCompletion.OnCompleted и ICriticalNotifyCompletion.UnsafeOnCompleted таким образом, чтобы делегат *r* вызывался не более одного раза. В противном случае поведение содержащей асинхронной функции будет неопределенным.

## Арифметические операторы

Операторы \*, /, %, + и – называются арифметическими операторами.

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

Если операнд арифметического оператора имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения тех операндов, которые имеют тип dynamic во время компиляции.

### Оператор произведения

Для операции вида x \* y, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Ниже перечислены стандартные операторы произведения. Все операторы вычисляют произведение x и y.

* Произведение целых чисел:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);

В контексте checked если произведение выходит за пределы диапазона типа результирующего значения, возникает исключение System.OverflowException. В контексте unchecked о переполнениях не сообщается, и все старшие биты, выходящие за пределы диапазона результирующего значения, отбрасываются.

* Произведение чисел с плавающей запятой

float operator \*(float x, float y);  
double operator \*(double x, double y);

Произведение вычисляется в соответствии с арифметическими правилами стандарта IEEE 754. В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и ошибок NaN. В таблице x и y являются положительными конечными значениями. z является результатом выражения x \* y. Если результат слишком велик для целевого типа, то z равно бесконечности. Если результат слишком мал для целевого типа, то z равно нулю.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +0 | –0 | +∞ | –∞ | NaN |
| –x | –z | +z | –0 | +0 | –∞ | +∞ | NaN |
| +0 | +0 | –0 | +0 | –0 | NaN | NaN | NaN |
| –0 | –0 | +0 | –0 | +0 | NaN | NaN | NaN |
| +∞ | +∞ | –∞ | NaN | NaN | +∞ | –∞ | NaN |
| –∞ | –∞ | +∞ | NaN | NaN | –∞ | +∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Произведение десятичных чисел.

decimal operator \*(decimal x, decimal y);

Если итоговое значение слишком велико для представления в формате decimal, создается исключение System.OverflowException. Если полученное значение слишком мало для представления в формате decimal, результат равен нулю. Масштаб результата до округления равен сумме масштабов двух операндов.

Произведение десятичных чисел эквивалентно использованию оператора произведения типа System.Decimal.

### Оператор деления

Для операции вида x / y, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Ниже перечислены стандартные операторы деления. Все операторы вычисляют частное x и y.

* Деление целых чисел:

int operator /(int x, int y);  
uint operator /(uint x, uint y);  
long operator /(long x, long y);  
ulong operator /(ulong x, ulong y);

Если значение правого операнда равно нулю, возникает исключение System.DivideByZeroException.

Деление округляет результат в сторону нуля. Таким образом, абсолютным значением результата является наибольшее целое число, меньшее или равное абсолютному значению частного двух операндов. Результат равен нулю или положителен, когда два операнда имеют один знак, и равен нулю или отрицателен, когда два операнда имеют противоположные знаки.

Если левый операнд является самым малым представимым значением int или long, а правый операнд равен –1, происходит переполнение. В контексте checked это приводит к исключению System.ArithmeticException (или его подклассу). В контексте unchecked возникновение исключения System.ArithmeticException (или его подкласс) или выдача сообщения о переполнении (с передачей результирующего значения равным левому операнду) определяется реализацией.

* Деление чисел с плавающей запятой:

float operator /(float x, float y);  
double operator /(double x, double y);

Частное вычисляется в соответствии с арифметическими правилами стандарта IEEE 754. В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и ошибок NaN. В таблице x и y являются положительными конечными значениями. z является результатом выражения x / y. Если результат слишком велик для целевого типа, то z равно бесконечности. Если результат слишком мал для целевого типа, то z равно нулю.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +∞ | –∞ | +0 | –0 | NaN |
| –x | –z | +z | –∞ | +∞ | –0 | +0 | NaN |
| +0 | +0 | –0 | NaN | NaN | +0 | –0 | NaN |
| –0 | –0 | +0 | NaN | NaN | –0 | +0 | NaN |
| +∞ | +∞ | –∞ | +∞ | –∞ | NaN | NaN | NaN |
| –∞ | –∞ | +∞ | –∞ | +∞ | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Деление десятичных чисел:

decimal operator /(decimal x, decimal y);

Если значение правого операнда равно нулю, возникает исключение System.DivideByZeroException. Если итоговое значение слишком велико для представления в формате decimal, создается исключение System.OverflowException. Если полученное значение слишком мало для представления в формате decimal, результат равен нулю. Масштаб результата равен минимальному масштабу, который позволит сохранить результат равный ближайшему к истинному математическому результату представимому десятичному числу.

Деление десятичных чисел эквивалентно использованию оператора деления типа System.Decimal.

### Оператор остатка

Для операции вида x % y, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Ниже перечислены стандартные операторы остатка. Все операторы вычисляют остаток деления x и y.

* Остаток для целых чисел:

int operator %(int x, int y);  
uint operator %(uint x, uint y);  
long operator %(long x, long y);  
ulong operator %(ulong x, ulong y);

Результатом выражения x % y является значение, получаемое в результате вычисления x – (x / y) \* y. Если y равно нулю, возникает исключение System.DivideByZeroException.

Если левый операнд является минимальным значением типа int или long, а правый операнд равен -1, то возникает исключение System.OverflowException. Если x % y не порождает исключения, x / y также не порождает исключения.

* Остаток для чисел с плавающей запятой:

float operator %(float x, float y);  
double operator %(double x, double y);

В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и ошибок NaN. В таблице x и y — положительные конечные значения. z является результатом x % y и вычисляется как x – n \* y, где n является максимальным возможным целым числом, меньшим или равным x / y. Этот метод вычисления остатка аналогичен методу, используемому для целых операндов, но отличается от определения по стандарту IEEE 754 (в котором n является целым числом, ближайшим к x / y).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | 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 |
| +∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| –∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Остаток для десятичных чисел:

decimal operator %(decimal x, decimal y);

Если значение правого операнда равно нулю, возникает исключение System.DivideByZeroException. Масштаб результат до округления равен большему масштабу из двух операндов, а знак результата (если он не равен нулю) равен знаку x.

Получение остатка для десятичных чисел эквивалентно использованию оператора остатка типа System.Decimal.

### Оператор сложения

Для операции вида x + y, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Ниже перечислены стандартные операторы сложения. Для числовых типов и типов перечислений стандартные операторы сложения вычисляют сумму двух операндов. Когда один или оба операнда имеют строковый тип, стандартные операторы сложения выполняют сцепление строковых представлений операндов.

* Сложение целых чисел:

int operator +(int x, int y);  
uint operator +(uint x, uint y);  
long operator +(long x, long y);  
ulong operator +(ulong x, ulong y);

В контексте checked, если сумма выходит за пределы диапазона типа результирующего значения, возникает исключение System.OverflowException. В контексте unchecked о переполнениях не сообщается, и все старшие биты, выходящие за пределы диапазона результирующего значения, отбрасываются.

* Сложение чисел с плавающей запятой:

float operator +(float x, float y);  
double operator +(double x, double y);

Сумма вычисляется в соответствии с арифметическими правилами стандарта IEEE 754. В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и ошибок NaN. В таблице x и y являются ненулевыми конечными значениями, а z является результатом выражения x + y. Если x и y имеют одинаковую величину, но противоположные знаки, то z равен положительному нулю. Если результат x + y слишком велик для представления в целевом типе, то z является бесконечным значением с таким же знаком, как и у x + y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | +∞ | –∞ | NaN |
| +0 | y | +0 | +0 | +∞ | –∞ | NaN |
| –0 | y | +0 | –0 | +∞ | –∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | +∞ | NaN | NaN |
| –∞ | –∞ | –∞ | –∞ | NaN | –∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Сложение десятичных чисел:

decimal operator +(decimal x, decimal y);

Если итоговое значение слишком велико для представления в формате decimal, создается исключение System.OverflowException. Масштаб результата до округления равен большему из масштабов двух операндов.

Получение суммы десятичных чисел эквивалентно использованию оператора сложения типа System.Decimal.

* Сложение элементов перечисления. В каждом типе перечисления неявно предоставляются следующие стандартные операторы, где E является типом перечисления, а U является базовым типом E:

E operator +(E x, U y);  
E operator +(U x, E y);

Во время выполнения операторы вычисляются в точности как (E)((U)x + (U)y).

* Сцепление строк:

string operator +(string x, string y);  
string operator +(string x, object y);  
string operator +(object x, string y);

Эти перегрузки бинарного оператора + выполняют объединение строк. Если один операнд при сцеплении строк равен null, то подставляется пустая строка. Иначе любой нестроковый аргумент преобразуется в свое строковое представление путем вызова виртуального метода ToString, наследуемого от типа object. Если метод ToString возвращает значение null, подставляется пустая строка.

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  
 }  
}

Результатом оператора сцепления строк является строка, состоящая из символов левого операнда, за которыми следуют символы правого операнда. Оператор сцепления строк никогда не возвращает значение null. При отсутствии памяти для размещения результирующей строки может возникать исключение System.OutOfMemoryException.

* Комбинация делегатов. В каждом типе делегата неявно предоставляется следующий стандартный оператор, где D имеет тип делегата:

D operator +(D x, D y);

Бинарный оператор + выполняет комбинацию делегатов, когда оба операнда имеют некоторый тип делегата D. (Если операнды имеют разные типы делегатов, возникает ошибка времени привязки.) Если первый операнд равен null, результат операции равен значению второго операнда (даже если оно также равно null). Иначе, если второй операнд равен null, то результатом операции является значение первого операнда. Иначе результатом операции является новый экземпляр делегата, который при вызове вызывает первый операнд, а затем второй операнд. Примеры комбинации операндов см. в разделах §7.8.5 и §15.4. Поскольку System.Delegate не является типом делегата, для него не определен operator +.

### Оператор вычитания

Для операции вида x – y, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Ниже перечислены стандартные операторы вычитания. Все операторы выполняют вычитание y из x.

* Вычитание целых чисел:

int operator –(int x, int y);  
uint operator –(uint x, uint y);  
long operator –(long x, long y);  
ulong operator –(ulong x, ulong y);

В контексте checked если разность выходит за пределы диапазона типа результирующего значения, возникает исключение System.OverflowException. В контексте unchecked о переполнениях не сообщается, и все старшие биты, выходящие за пределы диапазона результирующего значения, отбрасываются.

* Вычитание чисел с плавающей запятой:

float operator –(float x, float y);  
double operator –(double x, double y);

Разность вычисляется в соответствии с арифметическими правилами стандарта IEEE 754. В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и ошибок NaN В таблице x и y являются ненулевыми конечными значениями, а z является результатом выражения x – y. Если x и y равны, z равно положительному нулю. Если результат x – y слишком велик для представления в целевом типе, то z является бесконечным значением с таким же знаком, как и у x – y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | –∞ | +∞ | NaN |
| +0 | –y | +0 | +0 | –∞ | +∞ | NaN |
| –0 | –y | –0 | +0 | –∞ | +∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | NaN | +∞ | NaN |
| –∞ | –∞ | –∞ | –∞ | –∞ | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Вычитание десятичных чисел.

decimal operator –(decimal x, decimal y);

Если итоговое значение слишком велико для представления в формате decimal, создается исключение System.OverflowException. Масштаб результата до округления равен большему из масштабов двух операндов.

Вычитание десятичных чисел эквивалентно использованию оператора вычитания типа System.Decimal.

* Вычитание элементов перечисления. В каждом типе перечисления неявно предоставляются следующий стандартный оператор, где E является типом перечисления, а U является базовым типом E:

U operator –(E x, E y);

Этот оператор вычисляется в точности как (U)((U)x – (U)y). Другими словами, оператор вычисляет разность между порядковыми значениями x и y, а типом результата является базовый тип перечисления.

E operator –(E x, U y);

Этот оператор вычисляется в точности как (E)((U)x – y). Другими словами, оператор вычитает значение из базового типа перечисления, выдавая значение перечисления.

* Удаление делегатов. В каждом типе делегата неявно предоставляется следующий стандартный оператор, где D имеет тип делегата:

D operator –(D x, D y);

Бинарный оператор – выполняет удаление делегатов, когда оба операнда имеют некоторый тип делегата D. Если операнды имеют разные типы делегатов, возникает ошибка времени привязки. Если первый операнд равен null, то результатом операции является null. Иначе, если второй операнд равен null, то результатом операции является значение первого операнда. Иначе оба операнда представляют списки вызова (§15.1) с одной или несколькими записями, а результатом является новый список вызова, состоящий из списка первого операнда, из которого удалены записи второго операнда, при условии что список второго операнда является соответствующим смежным подсписком списка первого операнда. (Для определения равенства подсписков выполняется сравнение соответствующих записей, как и для оператора равенства делегатов (§7.10.8).) Иначе результатом является значение левого операнда. В ходе этого процессе списки операндов не меняются. Если список второго операнда соответствует нескольким подспискам последовательных записей в списке первого операнда, то самый правый совпадающий подсписок последовательных записей удаляется. Если после удаления получается пустой список, то результатом является null. Пример:

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  
 }  
}

## Операторы сдвига

Операторы << и >> используются для выполнения операций побитового сдвига.

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

Если операнд выражения-сдвига имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения тех операндов, которые имеют тип dynamic во время компиляции.

Для операции вида x << count или x >> count, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки бинарного оператора (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

При объявлении перегруженного оператора сдвига тип первого операнда всегда должен быть классом или структурой, в которой находится объявление оператора, а второй операнд должен иметь тип int.

Ниже перечислены стандартные операторы сдвига.

* Сдвиг влево:

int operator <<(int x, int count);  
uint operator <<(uint x, int count);  
long operator <<(long x, int count);  
ulong operator <<(ulong x, int count);

Оператор << выполняет сдвиг значения x влево на определенное число битов и вычисляется, как указано ниже.

Старшие биты вне диапазона типа результирующего значения x отбрасываются, оставшиеся биты сдвигаются влево, а пустые позиции младших битов заполняются нулями.

* Сдвиг вправо

int operator >>(int x, int count);  
uint operator >>(uint x, int count);  
long operator >>(long x, int count);  
ulong operator >>(ulong x, int count);

Оператор >> выполняет сдвиг значения x вправо на определенное число битов и вычисляется, как указано ниже.

Если x имеет тип int или long, то младшие биты x отбрасываются, оставшиеся биты сдвигаются вправо, а пустые позиции старших битов заполняются нулями, если x является неотрицательным числом, и заполняются единицами, если x является отрицательным числом.

Если x имеет тип uint или ulong, младшие биты x отбрасываются, оставшиеся биты сдвигаются вправо, а пустые позиции старших битов заполняются нулями.

Для стандартных операторов число сдвигаемых битов вычисляется следующим образом.

* Если x имеет тип int или uint, размер сдвига задается пятью младшими битами значения count. Другими словами, величина сдвига вычисляется от значения число count & 0x1F.
* Если x имеет тип long или ulong, размер сдвига задается шестью младшими битами значения count. Другими словами, величина сдвига вычисляется от значения число count & 0x3F.

Если результирующая величина сдвига равна нулю, то операторы сдвига просто возвращают значение x.

Операции сдвига никогда не вызывают переполнения и дают одинаковые результаты в контекстах checked и unchecked.

Если левый операнд оператора >> имеет целый тип со знаком, оператор выполняет арифметический сдвиг вправо, когда значение самого важного бита (бита знака) операнда распространяется на пустые позиции старших битов. Если левый операнд оператора >> имеет целый тип без знака, оператор выполняет логический сдвиг вправо, когда пустые позиции старших битов всегда заполняются нулями. Для выполнения обратной операции выведенной из типа операнда можно использовать явное приведение типов. Например, если x является переменной типа int, то операция unchecked((int)((uint)x >> y)) выполняет логический сдвиг x вправо.

## Операторы отношения и проверки типа

Операторы ==, !=, <, >, <=, >=, is и as называются операторами отношения и проверки типа.

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

Оператор is описывается в разделе §7.10.10, а оператор as — в разделе §7.10.11.

Операторы ==, !=, <, >, <= и >= являются операторами сравнения.

Если операнд оператора сравнения имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения тех операндов, которые имеют тип dynamic во время компиляции.

Для операции вида x op y, где op является оператором сравнения, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Стандартные операторы сравнения описываются в следующих разделах. Все стандартные операторы сравнения возвращают результат с типом bool, как показано в следующей таблице.

|  |  |
| --- | --- |
| **Операция** | **Результат** |
| x == y | Значение true, если значение x равно значению y; в противном случае — значение false. |
| x != y | Значение true, если значение x не равно значению y; в противном случае — значение false. |
| x < y | Значение true, если x меньше чем y, в противном случае — значение false. |
| x > y | Значение true, если x больше чем y, в противном случае — значение false. |
| x <= y | Значение true, если x меньше или равно y, в противном случае — значение false. |
| x >= y | Значение true, если x больше или равно y, в противном случае — значение false. |

### Операторы сравнения целых чисел

К стандартным операторам сравнения целых чисел относятся:

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);

Каждый из этих операторов сравнивает числовые значения двух целых операндов и возвращает значение bool, которое указывает, является ли результатом соответствующего отношения значение true или false.

### Операторы сравнения чисел с плавающей запятой

К стандартным операторам сравнения чисел с плавающей запятой относятся:

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);

Эти операторы сравнивают операнды в соответствии с правилами из стандарта IEEE 754.

* Если какой-либо из операторов не является числом (NaN), то результатом всех операторов является false, за исключением оператора !=, для которого результатом является true. Для любых двух операндов выражение x != y всегда дает такой же результат, что и выражение !(x == y). Однако когда один или оба операнда не являются числами (NaN), операторы <, >, <= и >= не дают такие же результаты, как логическое отрицание обратного оператора. Например, если x или y является не числом (NaN), то x < y имеет значение false, однако !(x >= y) имеет значение true.
* Когда оба операнда являются числами (NaN), значения двух операндов с плавающей запятой в операторах сравниваются в соответствии со следующим порядком

–∞ < –max < ... < –min < –0.0 == +0.0 < +min < ... < +max < +∞

где min и max являются самым малым и самым большим положительными значениями, которые можно представить в данном формате с плавающей запятой. Главным результатом такого порядка является следующее.

* Отрицательные и положительные нулевые значения считаются равными.
* Отрицательное бесконечное значение считается меньше всех остальных значений, но равным другому отрицательному бесконечному значению.
* Положительное бесконечное значение считается больше всех остальных значений, но равным другому положительному бесконечному значению.

### Операторы сравнения десятичных чисел

К стандартным операторам сравнения десятичных чисел относятся:

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);

Каждый из этих операторов сравнивает числовые значения двух десятичных операндов и возвращает значение bool, которое указывает, является ли результатом соответствующего отношения значение true или false. Каждый оператор сравнения десятичных чисел эквивалентен соответствующему оператору отношения или равенства типа System.Decimal.

### Логические операторы равенства

К стандартным логическим операторам равенства относятся:

bool operator ==(bool x, bool y);

bool operator !=(bool x, bool y);

Результат == равен значению true, если значения x и y одновременно равны true или если значения x и y одновременно равны false. В противном случае результатом будет false.

Результат != равен значению false, если значения x и y одновременно равны true или если значения x и y одновременно равны false. В противном случае результатом будет true. Если операнды имеют тип bool, оператор != дает такой же результат, как и оператор ^.

### Операторы сравнения значений перечисления

Каждый тип перечисления неявно предоставляет следующие стандартные операторы сравнения:

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);

Результат вычисления x op y, где x и y являются выражениями с типом перечисления E базового типа U, а op является одним из операторов сравнения, в точности равен значению ((U)x) op ((U)y). Другими словами, операторы сравнения значений типа перечисления просто сравнивают базовые целые значения двух операндов.

### Операторы равенства значений ссылочного типа

К стандартным операторам равенства значений ссылочного типа относятся:

bool operator ==(object x, object y);

bool operator !=(object x, object y);

Операторы возвращают результат сравнения двух ссылок на идентичность.

Поскольку стандартные операторы равенства значений ссылочного типа принимают операнды типов object, они применимы ко всем типам, в которых не объявляются применимые члены operator == и operator !=. И наоборот, любые применимые пользовательские операторы равенства скрывают стандартные операторы равенства значений ссылочного типа.

Для стандартных операторов равенства значений ссылочного типа должно выполняться одно из следующих условий.

* Оба операнда являются значениями типа, известного как ссылочный, или литералом null. Кроме того, из типа одного операнда в тип другого существует стандартное неявное преобразование (§6.2.4).
* Один операнд имеет значение типа T, где T является параметром типа, а второй операнд является литералом null. Кроме того, у T нет ограничений типа значения.

Если не выполняется какое-либо из этих требований, возникает ошибка времени привязки. Основные следствия из этих правил таковы.

* Если известно, что во время привязки две ссылки отличаются, то использование стандартных операторов равенства значений ссылочного типа для их сравнения вызывает ошибку времени привязки. Например, если типы операндов во время привязки — это типы классов A и B и если ни A, ни B не является производным от другого класса, то эти два операнда не смогут ссылаться на один объект. Следовательно, операция вызовет ошибку времени привязки.
* Стандартные операторы равенства значений ссылочного типа не позволяют сравнивать операнды с типом значения. Поэтому нельзя сравнивать значения типа структуры, если только в этом типе не объявляются собственные операторы сравнения.
* Стандартные операторы равенства значений ссылочного типа никогда не вызывают выполнение операций упаковки для своих операндов. Выполнение таких операций упаковки будет бессмысленным, поскольку ссылки на создаваемые упакованные экземпляры обязательно будут отличаться от всех остальных ссылок.
* Если операнд с типом параметра типа T сравнивается со значением null и тип T во время выполнения является типом значения, то результат сравнения равен false.

В следующем примере проверяется, имеет ли аргумент типа параметра типа без ограничений значение null.

class C<T>  
{  
 void F(T x) {  
 if (x == null) throw new ArgumentNullException();  
 ...  
 }  
}

Конструкция x == null разрешена, даже если T может представлять тип значения, и результат просто равен false, когда T имеет тип значения.

Для операции вида x == y или x != y если существует применимый operator == или operator !=, то по правилам разрешения перегрузки операторов (§7.3.4) будет выбран этот оператор вместо стандартного оператора равенства значений ссылочного типа. Однако всегда можно выбрать стандартный оператор равенства значений ссылочного типа, выполнив явное приведение одного или обоих операндов к типу object. Пример:

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);  
 }  
}

производятся следующие выходные данные

True  
False  
False  
False

Переменные s и t относятся к разным экземплярам string, содержащим одинаковые символы Первое сравнение дает значение True, потому что когда оба операнда имеют тип string выбирается стандартный оператор равенства значений ссылочного типа (§7.10.7). В оставшихся операциях сравнения получается значение False, потому что когда один или оба операнда имеют тип object, выбирается стандартный оператор равенства значений ссылочного типа.

Обратите внимание, что указанная процедура не применима для типов значения. Пример:

class Test  
{  
 static void Main() {  
 int i = 123;  
 int j = 123;  
 System.Console.WriteLine((object)i == (object)j);  
 }  
}

выдается результат False, потому что при приведении типов создаются ссылки на два разных экземпляра упакованных значений int.

### Операторы равенства строк

К стандартным операторам равенства строк относятся:

bool operator ==(string x, string y);

bool operator !=(string x, string y);

Два значения типа string считаются равными, когда истинно одно из следующих условий:

* оба значения равны null;
* оба значения являются непустыми ссылками на экземпляры строк с одинаковой длиной и идентичными символами в каждой позиции.

Операторы равенства строк сравнивают значения строк, а не ссылки на строки. Когда два разных экземпляра строк содержат одинаковую последовательность символов, значения строк равны, но ссылки отличаются. Как описано в разделе §7.10.6, для сравнения ссылок на строки вместо строковых значений можно использовать операторы равенства значений ссылочного типа.

### Операторы равенства делегатов

Каждый тип делегата неявно предоставляет следующие стандартные операторы сравнения:

bool operator ==(System.Delegate x, System.Delegate y);

bool operator !=(System.Delegate x, System.Delegate y);

Два делегата считаются равными, если выполняется одно из следующих условий.

* Если один из экземпляров делегата равен null, то они равны, только они оба равны null.
* Если делегаты имеют разный тип во время выполнения, то они не могут быть равны.
* Если у обоих экземпляров делегата есть список вызова (§15.1), то они равны, только если их списки вызова имеют одну длину и каждая запись в списке вызова одного делегата равна (в соответствии с определением ниже) соответствующей записи по порядку в списке вызова другого делегата.

Равенство записей списка вызова определяется следующими правилами.

* Если две записи списка вызова ссылаются на один статический метод, то записи равны.
* Если две записи списка вызова ссылаются на один нестатический метод одного целевого объекта (определяется операторами равенства ссылочных значений), то записи равны.
* Записи списка вызова, полученные при вычислении семантически идентичных выражений анонимных функций с одинаковым (возможно пустым) набором захваченных внешних экземпляров переменных могут (но не обязательно) быть равны.

### Операторы равенства и значение NULL

Операторы == и != допускают, что один операнд был значением обнуляемого типа, а второй литералом null, даже если для операции не существует стандартного или пользовательского оператора (обычного или с нулификацией).

Для операции одного из видов

x == null null == x x != null null != x

где x является выражением обнуляемого типа, если при разрешении перегрузки операторов (§7.2.4) не удается найти применимый оператор, то результат вычисляется на основании свойства HasValue для x. В частности, первые два вида операции преобразуются в!x.HasValue, а два последних вида преобразуются в x.HasValue.

### Оператор is

Оператор is используется для динамической проверки, можно ли сравнивать тип объекта во время выполнения с указанным типом. Результат операции E is T, где E — это выражение, а T — тип, является логическим значением, которое указывает, можно ли провести преобразование E в тип T с помощью преобразования ссылки, преобразования упаковки или преобразования отмены упаковки. После замены всех параметров типа на аргументы типа операция вычисляется следующим образом.

* Если E является анонимной функцией, возникает ошибка времени компиляции
* Если E является группой методов или литералом null или если E имеет ссылочный или обнуляемый и значение E равно NULL, то результат равен false.
* Иначе пусть D обозначает динамический тип E следующим образом.
* Если E имеет ссылочный тип, D является типом времени выполнения для экземпляра, на который ссылается E.
* Если E имеет обнуляемый тип, D является базовым типом этого обнуляемого типа.
* Если E имеет необнуляемый тип значения, D имеет тип E.
* Результат операции зависит от D и T следующим образом.
* Если T имеет ссылочный тип, то результат равен true, если D и T имеют одинаковый тип, если D имеет ссылочный тип и существует неявное преобразование ссылки из D в T или если D имеет тип значения и существует преобразование упаковки из D в T.
* Если T имеет обнуляемый тип, результат равен true, если D является базовым типом T.
* Если T является необнуляемым типом значения, результат равен true, если D и T имеют одинаковый тип
* Иначе результат равен false.

Обратите внимание, что пользовательские преобразования в операторе is не учитываются.

### Оператор as

Оператор as используется для явного преобразования значения в указанный ссылочный тип или обнуляемый тип. В отличие от выражения приведения типа (§7.7.6) оператор as никогда не вызывает исключения. Вместо этого, если указанное преобразование невозможно, возвращается значение null.

В операции вида E as T E должно быть выражением, а T должен быть ссылочным типом, параметром типа, который является ссылочным типом, или обнуляемым типом. Кроме того, должно выполняться, по крайней мере, одно из следующих условий. В противном случае будет возникать ошибка времени компиляции.

* Из E в T существует преобразование идентификации (§6.1.1), неявное преобразование обнуляемых типов (§6.1.4), неявное преобразование ссылочных типов (§6.1.6), преобразование упаковки (§6.1.7), явное преобразование обнуляемых типов (§6.2.3), явное преобразование ссылочных типов (§6.2.4) и преобразование отмены упаковки (§6.2.5).
* Тип E или T является открытым.
* E является литералом null.

Если тип времени компиляции E не является dynamic, операция E as T дает такой же результат, как и

E is T ? (T)(E) : (T)null

за исключением того, что E вычисляется только один раз. Компилятор может проводить оптимизацию операции E as T, чтобы при ее обработке выполнялась только одна динамическая проверка типа в отличие от двух динамических проверок типа в расширении выше.

Если тип времени компиляции E является dynamic, то, в отличие от оператора приведения, оператор as не имеет динамической привязки (§7.2.2). Таким образом, развертывание в данном случае будет следующим:

E is T ? (T)(object)(E) : (T)null

Обратите внимание, что некоторые преобразования, например пользовательские преобразования нельзя использовать с оператором as и вместо них следует использовать выражения приведения типа.

В этом примере

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   
 }  
}

параметр T для G имеет ссылочный тип, поскольку у него есть ограничение класса. Напротив, параметр типа U для H не имеет ссылочного типа, поэтому использование оператора as в H запрещено.

## Логические операторы

Операторы &, ^ и | называются логическими операторами.

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

Если операнд логического оператора имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения тех операндов, которые имеют тип dynamic во время компиляции.

Для операции вида x op y, где op является одним из логических операторов, чтобы выбрать конкретную реализацию оператора, применяется разрешение перегрузки (§7.3.4). Операнды преобразуются в типы параметров выбранного оператора, а тип результата является типом возвращаемого значения этого оператора.

Стандартные логические операторы описываются в следующих разделах.

### Логические операторы для целых чисел

К стандартным логическим операторам для целых чисел относятся:

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);

Оператор & выполняет логическую побитовую операцию AND для двух операндов, оператор | выполняет логическую побитовую операцию OR для двух операндов, а оператор ^ выполняет логическую побитовую операцию исключающего OR для двух операндов. Эти операции не порождают переполнения.

### Логические операторы для перечислений

Каждый тип перечисления E неявно предоставляет следующие стандартные логические операторы:

E operator &(E x, E y);  
E operator |(E x, E y);  
E operator ^(E x, E y);

Результат вычисления x op y, где x и y являются выражениями с типом перечисления E базового типа U, а op является одним из логических операторов, в точности равен значению (E)((U)x op (U)y). Другими словами, логические операторы для перечислений просто выполняют логические операции над базовым типом двух операндов.

### Логические операторы

К стандартным логическим операторам относятся:

bool operator &(bool x, bool y);

bool operator |(bool x, bool y);

bool operator ^(bool x, bool y);

Результат выражения x & y равен true, если значения x и y одновременно равны true. В противном случае результатом будет false.

Результат выражения x | y равен true, если значение x или y равно true. В противном случае результатом будет false.

Результат выражения x ^ y равен значению true, если значение x равно true, а значение y равно false или значение x равно false, а значение y равно true. В противном случае результатом будет false. Когда операнды имеют тип bool, оператор ^ дает такой же результат, как и оператор !=.

### Обнуляемые логические операторы

Обнуляемый логический тип bool? может представлять три значения: true, false и null. По сути, он аналогичен типу из трех значений, используемому в логических выражениях в SQL. Чтобы обеспечить согласованность результатов операторов & и | с операндами типа bool? с троичной логикой SQL, предоставляются следующие стандартные операторы:

bool? operator &(bool? x, bool? y);

bool? operator |(bool? x, bool? y);

В следующей таблице перечисляются результаты данных операторов для всех комбинаций значений true, false и null.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | x & y | x | y |
| true | true | true | true |
| true | false | false | true |
| true | null | null | true |
| false | true | false | true |
| false | false | false | false |
| false | null | false | null |
| null | true | null | true |
| null | false | false | null |
| null | null | null | null |

## Условные логические операторы

Операторы && и || называются условными логическими операторами. Они также называются логическими операторами «краткой записи».

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

Операторы && и || являются условными версиями операторов & и |.

* Операция x && y соответствует операции x & y, за исключением того, что y вычисляется, только если x не равен false.
* Операция x || y соответствует операции x | y, за исключением того, что y вычисляется, только если x не равен true.

Если операнд условного логического оператора имеет тип dynamic во время компиляции, то он динамически связан (§7.2.2). В этом случае тип времени компиляции выражения — dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения при использовании типа времени выполнения тех операндов, которые имеют тип dynamic во время компиляции.

Операция вида x && y или x || y обрабатывается с применением разрешения перегрузки (§7.3.4), как если бы операция записывалась в виде x & y или x | y. Тогда

* Если при разрешении перегрузки не удается найти один наилучший оператор или выбирается один из стандартных логических операторов для целых чисел, то возникает ошибка времени привязки.
* Иначе, если выбранный оператор является одним из стандартных логических операторов (§7.11.3) или обнуляемых логических операторов (§7.11.4), то операция выполняется, как описано в разделе §7.12.1.
* Иначе выбранный оператор является пользовательским оператором, и операция выполняется, как описано в разделе §7.12.2.

Напрямую перегружать условные логические операторы нельзя. Однако поскольку условные логические операторы обрабатываются через обычные логические операторы, перегрузки обычных логических операторов с определенными ограничениями также считаются перегрузками условных логических операторов. Это описано далее в §7.12.2.

### Логические условные операторы

Если операнды && или || имеют тип bool или типы, в которых не определены применимые операторы operator & или operator |, но определены неявные преобразования в bool, то операция выполняется следующим образом.

* Операция x && y вычисляется как x ? y : false. Другими словами, сначала вычисляется x и затем преобразуется в тип bool. Затем, если x равен true, y вычисляется и преобразуется в тип bool и становится результатом операции. Иначе результат операции равен false.
* Операция x || y вычисляется как x ? true : y. Другими словами, сначала вычисляется x и затем преобразуется в тип bool. Затем, если x равно true, результат операции равен true. Иначе y вычисляется и преобразуется в тип bool и становится результатом операции.

### Пользовательские условные логические операторы

Если операнды && или || имеют типы, в которых объявляются применимые пользовательские операторы operator & или operator |, то должны выполняться оба следующих условия, где T является типом, в котором объявляется выбранный оператор.

* Типом возвращаемого значения и типом каждого параметра выбранного оператора должен быть T. Другими словами, оператор должен вычислять операцию логического AND или логического OR для двух операндов типа T и должен возвращать результат типа T.
* T должен содержать объявления операторов operator true и operator false.

Если какое-либо из этих условий не выполняется, возникает ошибка времени привязки. Иначе операция && или || вычисляется путем объединения пользовательских операторов operator true и operator false с выбранным пользовательским оператором:

* Операция x && y вычисляется как T.false(x) ? x : T.&(x, y), где T.false(x) является вызовом оператора operator false, объявленного в T, а T.&(x, y) является вызовом выбранного оператора operator &. Другими словами, сначала вычисляется x и для результата вызывается operator false, чтобы определить, имеет ли x значение false. Затем, если x имеет значение false, результатом операции становится значение, ранее вычисленное для x. Иначе вычисляется y и выбранный оператор operator & вызывается для ранее вычисленного значения для x и вычисленного значения y, чтобы получить результат операции.
* Операция x || y вычисляется как T.true(x) ? x : T.|(x, y), где T.true(x) является вызовом оператора operator true, объявленного в T, а T.|(x, y) является вызовом выбранного оператора operator |. Другими словами, сначала вычисляется x и для результата вызывается operator true, чтобы определить, имеет ли x значение true. Затем, если x имеет значение true, результатом операции становится значение, ранее вычисленное для x. Иначе вычисляется y и выбранный оператор operator | вызывается для ранее вычисленного значения для x и вычисленного значения y, чтобы получить результат операции.

В обеих этих операциях выражение в x вычисляется только один раз, а выражение в y либо не вычисляется, либо вычисляется ровно один раз.

Пример типа, реализующего операторы operator true и operator false, см. в разделе §11.4.2.

## Оператор слияния с NULL

Оператор ?? называется оператором слияния с NULL.

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

В выражении слияния с NULL вида a ?? b требуется, чтобы a имело обнуляемый или ссылочный тип. Если a не равно значению NULL, результатом выражения a ?? b является a; в противном случае результатом является b. В операции вычисление b происходит, только если a равно NULL.

Оператор слияния с null имеет правую ассоциативность, что означает, что операции группируются справа налево. Например, выражение вида a ?? b ?? c вычисляется как a ?? (b ?? c). В общем случае выражение вида E1 ?? E2 ?? ... ?? EN возвращает первый операнд, не равный NULL, или NULL, если все операнды равны NULL.

Тип выражения a ?? b зависит от того, какие неявные преобразования доступны с операндами. В порядке предпочтения тип a ?? b равен A0, A или B, где A является типом a (при условии, что у a есть тип), B является типом b (при условии, что у b есть тип), а A0 является базовым типом A, если A является обнуляемым типом, либо равен типу A в противном случае. В частности, a ?? b обрабатывается следующим образом.

* Если A существует и не является обнуляемым типом или ссылочным типом, возникает ошибка времени компиляции.
* Если b является динамическим выражением, то тип результата будет dynamic. Во время выполнения сначала вычисляется a. Если a не равно NULL, для a выполняется преобразование в динамический тип, и это становится результатом. Иначе вычисляется b и это становится результатом.
* Иначе, если A существует и является обнуляемым типом и существует неявное преобразование из b в A0, то типом результата будет A0. Во время выполнения сначала вычисляется a. Если a не равно NULL, для a выполняется снятие упаковки до типа A0 и это становится результатом. Иначе вычисляется b и преобразуется в тип A0 и это становится результатом.
* Иначе, если существует A и существует неявное преобразование из b в A, то типом результата будет A. Во время выполнения сначала вычисляется a. Если a не равно NULL, a становится результатом. Иначе вычисляется b и преобразуется в тип A и это становится результатом.
* Иначе, если b имеет тип B и существует неявное преобразование из a в B, то типом результата будет B. Во время выполнения сначала вычисляется a. Если a не равно NULL, для a выполняется снятие упаковки до типа A0 (если только A существует и является обнуляемым), оно преобразуется в тип B и это становится результатом. Иначе вычисляется b и это становится результатом.
* Иначе a и b являются несовместимыми и возникает ошибка времени компиляции.

## Условный оператор

Оператор ?: называется условным оператором. Иногда его также называют тернарным оператором.

conditional-expression:  
null-coalescing-expression  
null-coalescing-expression ? expression : expression

В условном выражении вида b ? x : y сначала вычисляется условие b. Затем, если b равно true, вычисляется x и это становится результатом операции. Иначе вычисляется y и это становится результатом операции. В условном выражении никогда не выполняется вычисление и x, и y.

Условный оператор имеет правую ассоциативность, что означает, что операции группируются справа налево. Например, выражение вида a ? b : c ? d : e вычисляется как a ? b : (c ? d : e).

Первый операнд оператора ?: должен быть выражением с типом, который можно неявно преобразовать в тип bool, или выражением типа, в котором реализован оператор operator true. Если не выполняется ни одно из этих требований, то возникает ошибка времени компиляции.

Второй и третий операнды, x и y, оператора ?: задают тип условного выражения.

* Если x имеет тип X, а y имеет тип Y, то
* Если из X в Y существует неявное преобразование (§6.1), а из Y в X не существует, то типом условного выражения является Y.
* Если из Y в X существует неявное преобразование (§6.1), а из X в Y не существует, то типом условного выражения является X.
* Иначе тип выражения определить нельзя, и возникает ошибка времени компиляции.
* Если только один из x и y имеет тип, и оба x и y могут быть неявно преобразованы в этот тип, то он является типом условного выражения.
* Иначе тип выражения определить нельзя, и возникает ошибка времени компиляции.

Во время выполнения обработка условного выражения вида b ? x : y включает следующие этапы.

* Сначала вычисляется b и определяется значение b типа bool:
* Если существует неявное преобразование из типа b в bool, то для получения значения типа bool выполняется это неявное преобразование.
* В противном случае для получения значения типа bool вызывается оператор operator true, определенный типом b.
* Если полученное на предыдущем этапе значение типа bool равно true, то x вычисляется и преобразуется в тип условного выражения и это становится результатом условного выражения.
* Иначе y вычисляется и преобразуется в тип условного выражения и это становится результатом условного выражения.

## Выражения анонимных функций

Анонимная функция — это выражение, представляющее собой подставляемое определение метода. Анонимная функция не имеет значения или тип сама по себе, но может быть преобразована в совместимый тип делегата или дерева выражения. Преобразование анонимной функции зависит от целевого типа преобразования. Если это тип делегата, то результатом преобразования является значение делегата, ссылающееся на метод, определяемый анонимной функцией. Если это тип дерева выражения, то результатом преобразования является дерево выражения, которое представляет структуру метода в виде структуры объекта.

По историческим причинам существует две синтаксические разновидности анонимных функций, а именно: лямбда-выражения и выражения анонимных методов. Практически для любых целей лямбда-выражения более конкретны и точны, чем выражения анонимных методов, которые остаются в языке для обеспечения обратной совместимости.

lambda-expression:  
async*opt* anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
async*opt* 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-list*opt* )

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-list*opt* )  
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

Оператор => имеет такой же приоритет, как и присваивание (=) и обладает правой ассоциативностью.

Анонимная функция с модификатором async является асинхронной функцией, которая следует правилам, описанным в разделе §10.14.

Тип параметров анонимной функции в виде лямбда-выражения может задаваться явно или неявно. В списке явно типизированных параметров тип каждого параметра указывается явно. В списке неявно типизированных параметров типы параметров выводятся из контекста, в котором находится анонимная функция, в частности, когда анонимная функция преобразуется в совместимый тип делегата или дерева выражения, типы параметров предоставляются этим типом (§6.5).

В анонимной функции с одним параметром с неявной типизацией в списке параметров можно опустить скобки. Другими словами запись анонимной функции вида

( param ) => expr

можно сократить до:

param => expr

Список параметров анонимной функции в виде выражения анонимного метода является необязательным. Если они заданы, то параметры должны быть явно типизированы Если они не задаются, то анонимную функцию можно преобразовать в делегат с любым списком параметров, не содержащих параметров out.

Тело блока анонимной функции достижимо (§8.1), если анонимная функция не расположена внутри недостижимого оператора.

Ниже приведены некоторые примеры анонимных функций.

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

Поведение лямбда-выражений и выражений анонимных методов совпадает за исключением следующих моментов:

* выражения анонимных методов позволяют опускать список параметров целиком, обеспечивая возможность преобразования в типы делегатов с любым списком параметров значения.
* лямбда-выражения позволяют опускать и выводить типы параметров, тогда как в выражениях анонимных методов типы параметров должны быть указаны явно.
* Тело лямбда-выражения может быть выражением или блоком оператора, тогда как тело выражения анонимного метода должно быть блоком оператора.
* Только для лямбда-выражений существуют преобразования в совместимые деревья выражений (§4.6).

### Сигнатуры анонимных функций

Необязательная подпись анонимной функции задает названия и (необязательно) имена формальных параметров анонимной функции. Область действия параметров анонимной функции — это тело анонимной функции. (§3.7) Вместе со списком параметров (если он предоставлен) тело анонимного метода задает пространство объявления (§3.3). Таким образом, если имя параметра анонимной функции совпадает с именем локальной переменной, локальной константы или параметра, область действия которого включает выражение анонимного метода или лямбда-выражение, то будет возникать ошибка времени компиляции.

Если у анонимной функции есть явная подпись анонимной функции, то набор совместимых типов делегатов и типов дерева выражения ограничивается теми, у которых имеются одинаковые параметры типа и модификаторы в том же порядке. В отличие от преобразования группы методов (§6.6) контравариантность типов параметров анонимных функций не поддерживается. Если у анонимной функции нет подписи анонимной функции, то набор совместимых типов делегатов и типов деревьев выражений ограничивается тем, у которых нет параметров out.

Обратите внимание, что сигнатура анонимной функции не может включать атрибуты или массив параметров. Тем не менее сигнатура анонимной функции может быть совместима с типом делегата, список параметров которого содержит массив параметров.

Обратите внимание, что преобразование в тип дерева выражения, даже если он совместим, может привести к сбою во время компиляции (§4.6).

### Тела анонимных функций

Тело (выражение или блок) анонимной функции должно удовлетворять следующим правилам.

* Если анонимная функция включает сигнатуру, то параметры, указанные в сигнатуре, должны быть доступны в теле. Если у анонимной функции нет сигнатуры, она может быть преобразована в тип делегата или тип выражения с параметрами (§6.5), но обращаться к параметрам в теле будет нельзя.
* Если в теле происходит обращение к параметру ref или out, то возникает ошибка времени компиляции, за исключением случаев, когда параметры ref или out указываются в подписи (если она есть) ближайшей включающей анонимной функции.
* Если типом this является тип структуры, то при обращении в теле к this возникает ошибка времени выполнения. Это так при явном доступе (как в случае this.x) и при неявном (ср. x, где x является членом экземпляра структуры). Это правило просто запрещает такой доступ и не влияет на то, будет ли при поиске члена найден член структуры.
* В теле можно обращаться к внешним переменным (§7.15.5) анонимной функции. При доступе к внешней переменной будет сформирована ссылка на экземпляр переменной, активной во время вычисления лямбда-выражения или выражения анонимного метода (§7.15.6).
* При наличии в теле оператора goto, break или continue, назначением которого является расположение вне тела или внутри тела содержащейся анонимной функции, будет возникать ошибка времени компиляции.
* Оператор return в теле возвращает управление из вызова ближайшей включающей анонимной функции, а не из включающей функции-члена. Выражение, указанное в операторе return, должно быть неявно преобразуемым в возвращаемый тип типа делегата или типа дерева выражений, в который преобразуется ближайшее включающее лямбда-выражение или выражение анонимного метода (§6.5).

Специально не указывается, существует ли способ выполнить блок анонимной функции, отличный от вычисления и вызова лямбда-выражения или выражения анонимного метода. В частности, компилятор может реализовать анонимную функцию, создав один или несколько именованных методов или типов. Имена любых таких созданных элементов должны иметь формат, зарезервированный для использования компилятором.

### Разрешение перегрузки

Анонимные функции в списке аргументов участвуют в выводе типа и разрешении перегрузки. Подробные правила см. в разделах §7.5.2 и §7.5.3.

В следующем примере демонстрируется результат использования анонимных функций при разрешении перегрузки

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;  
 }  
}

Класс ItemList<T> содержит два метода Sum. У каждого метода есть аргумент selector, который извлекает значение для суммирования из списка элементов. Извлеченное значение может иметь тип int или double, и сумма в результате также будет иметь тип int или double.

Методы Sum можно, например, использовать для вычисления сумм из списка строк с деталями заказов.

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);  
 ...  
}

В первом вызове orderDetails.Sum оба метода Sum применимы, поскольку анонимная функция d => d.UnitCount совместима и с Func<Detail,int> и с Func<Detail,double>. Однако при разрешении перегрузки будет выбран первый метод Sum, потому что преобразование в Func<Detail,int> оказывается лучше преобразования в Func<Detail,double>.

Во втором вызове orderDetails.Sum применим только второй метод Sum, потому что анонимная функция d => d.UnitPrice \* d.UnitCount возвращает значение типа double. Поэтому в данном вызове при разрешении перегрузки будет выбран второй метод Sum.

### Анонимные функции и динамическая привязка

Анонимная функция не может быть получателем, аргументом или операндом динамически привязанной операции.

### Внешние переменные

Любая локальная переменная, параметр значения или массив параметров, область действия которого включает лямбда-выражение или выражение анонимного метода, называется внешней переменной анонимной функции. В функции-члене экземпляра для класса значение this считается параметром значения и является внешней переменно любой анонимной функции, содержащейся в функции-члене.

#### Захваченные внешние переменные

Когда анонимная функция ссылается на внешнюю переменную, внешнюю переменную называют захваченной анонимной функцией. Обычно срок жизни локальной переменной ограничен выполнением блока или оператора, с которым она связана (§5.1.7). Однако срок жизни захваченной внешней переменной увеличивается по крайней мере до того, как делегат или дерево выражения, созданное из анонимной функции, становится объектом для процесса сборки мусора.

В этом примере

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());  
 }  
}

локальная переменная x захватывается анонимной функцией и срок жизни x увеличивается по крайней до тех пор, пока делегат, возвращенный из F, не станет объектом для процесса сборки мусора (что произойдет только в самом конце программы). Поскольку каждый вызов анонимной функции работает с одним экземпляром x, пример имеет следующий вывод:

1  
2  
3

Когда локальная переменная или параметр значения захватывается анонимной функцией, локальная переменная или параметр больше не считается фиксированной переменной (§18.3), а вместо этого считается перемещаемой переменной. Поэтому в любом небезопасном (unsafe) коде, который получает адрес захваченной внешней переменной, сначала необходимо использовать оператор fixed, чтобы зафиксировать переменную.

Заметьте, в отличие от незахваченной переменной, захваченная локальная переменная может быть одновременно использоваться в нескольких потоках выполнения.

#### Создание экземпляров локальных переменных

Считается, что для локальной переменной создается экземпляр, когда выполнение входит в область действия переменной. Например, при вызове следующего метода экземпляр локальной переменной x создается и инициализируется три раза, один раз для каждой итерации цикла.

static void F() {  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 ...  
 }  
}

Однако если перенести объявления x за пределы цикла, то экземпляр x будет создаваться только один раз.

static void F() {  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 ...  
 }  
}

Если локальная переменная не захвачена, то не существует способа точно отследить, сколько раз создавались ее экземпляры, потому что сроки жизни экземпляров не связаны. Для каждого экземпляра можно просто использовать одно место хранения. Однако когда локальная переменная захватывается анонимной функцией, результат создания экземпляра становится очевидным.

Пример:

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();  
 }  
}

дает на выходе:

1  
3  
5

Однако при переносе объявления x за пределы цикла:

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;  
}

вывод имеет вид:

5  
5  
5

Если в цикле for объявляется переменная итерации, то сама эта переменная считается объявленной за пределами цикла. Таким образом, если изменить пример, чтобы захватывать саму переменную итерации

static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 result[i] = () => { Console.WriteLine(i); };  
 }  
 return result;  
}

то будет захвачен только один экземпляр переменной итерации. В этом случае вывод имеет вид

3  
3  
3

Делегаты анонимных функций могут иметь некоторые общие захваченные переменные и в то же время иметь отдельные экземпляры других переменных. Например, если F изменить на

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;  
}

то три делегата будут захватывать один экземпляр x, но разные экземпляры y. Вывод будет иметь вид:

1 1  
2 1  
3 1

Отдельные анонимные функции могут захватывать один экземпляр внешней переменной. В примере:

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());  
 }  
}

две анонимные функции захватывают один экземпляр локальной переменной x и поэтому они могут "взаимодействовать" через эту переменную. Вывод в данном примере имеет вид:

5  
10

### Вычисление выражений анонимных функций

Анонимная функция F всегда должна преобразовываться в тип делегата D или тип дерева выражения E либо напрямую, либо с помощью выполнения выражения создания делегата new D(F). Это преобразование определяет результат анонимной функции, как описано в разделе §6.5.

## Выражения запросов

Выражения запросов представляют собой подмножество синтаксических конструкций языка для запросов, похожих запросы иерархических или реляционных языков запросов, таких как SQL и XQuery.

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

Выражение запроса начинается с предложения from и заканчивается предложением select или group. После начального предложения from могут находиться (или отсутствовать) предложения from, let, where, join или orderby. Каждое предложение from является генератором, предлагающим переменную диапазона, которая включает элементы последовательности. Каждое предложение let вводит переменную диапазона, которая представляет значение, вычисляемое с помощью предыдущих переменных диапазона. Каждое предложение where является фильтром для исключения элементов из результата. Каждое предложение join сравнивает указанные ключи исходной последовательности с ключами другой последовательности, выдавая совпадающие пары. Каждое предложение orderby изменят порядок элементов в соответствии с указанными критериями. Конечное предложение select или group задает формат результата в виде переменных диапазона. И наконец, предложение into можно использовать для "склеивания" запросов, рассматривая результаты одного запроса в качестве генератора для последующего запроса.

### Неоднозначность в выражениях запросов

Выражения запросов содержат определенное число «контекстных ключевых слов», то есть идентификаторов, у которых есть особое значение в конкретном контексте. В частности, такими ключевыми словами могут быть from, where, join, on, equals, into, let, orderby, ascending, descending, select, group и by. Чтобы не допустить появления неоднозначности в выражениях запросов, вызванной смешанным использованием этих идентификаторов в виде ключевых слов и простых имен, при появлении в любом месте выражения запроса такие идентификаторы считаются ключевыми словами.

В этом смысле выражением запроса является любое выражение, которое начинается со строки "from идентификатор", за которым следует любая лексема кроме ";", "=" или ",".

Чтобы использовать эти слова в качестве идентификаторов в выражении запроса, перед ними можно указать "@" (§2.4.2).

### Перевод выражений запросов

В языке C# не задается семантика выполнения выражений запросов. Вместо этого выражения запросов переводятся в вызовы методов, которые соответствуют шаблону выражения запроса (§7.16.3). В частности, выражения запросов преобразуются в вызовы методов с именами Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy и Cast. Эти методы должны иметь определенные сигнатуры и типы результатов, как описано в разделе §7.16.3. Эти методы могут быть методами экземпляра запрашиваемого объекта или методами расширения, внешними для объекта, и в них может реализовываться фактическое выполнение запроса.

Перевод из выражения запроса в вызов метода состоит в синтаксическом сопоставлении, которое происходит до проведения каких-либо привязок или разрешения перегрузки. Перевод гарантированно является синтаксически корректным, однако семантическая корректность кода C# не гарантируется. После перевода выражений запросов получившиеся вызовы методов обрабатываются как обычные вызовы методов. При этом также могут обнаруживаться ошибки, например, если методы не существуют, если аргументы имеют неправильные типы или если методы являются универсальными и вывод типов завершается сбоем.

Обработка выражения запроса повторяется с применением следующих переводов до тех пор, пока не удается выполнить дальнейшее сокращение. Переводы приведены в порядке применения: для каждого раздела подразумевается, что переводы в предыдущих разделах были выполнены полностью, и по завершении раздел не будет пересматриваться при обработке такого же выражения запроса.

Присвоение для переменных диапазона не разрешено в выражениях запросов. Однако при реализации C# разрешается не во всех случаях выполнять это ограничение, поскольку иногда это становится невозможным для представленной здесь синтаксической схемы перевода.

При некоторых переводах вставляются переменные диапазона с прозрачными идентификаторами, обозначенными \*. Особые свойства прозрачных идентификаторов рассматриваются в разделе §7.16.2.7.

#### Предложения select и groupby с продолжениями

Выражение запроса с продолжением

from … into x …

переводится в

from x in ( from … ) …

В переводах в следующих разделах предполагается, что в запросах нет продолжений into.

Пример:

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

переводится в

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

конечный перевод имеет вид

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

#### Явные типы переменных диапазона

Предложение from, явно указывающее тип переменной диапазона

from T x in e

переводится в

from x in ( e ) . Cast < T > ( )

Предложение join, явно указывающее тип переменной диапазона

join T x in e on k1 equals k2

переводится в

join x in ( e ) . Cast < T > ( ) on k1 equals k2

В переводах в следующих разделах предполагается, что в запросах нет явных типов переменных диапазона.

Пример:

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

переводится в

from c in customers.Cast<Customer>()  
where c.City == "London"  
select c

конечный перевод имеет вид

customers.  
Cast<Customer>().  
Where(c => c.City == "London")

Явные типы переменных диапазона удобно использовать при запросе к коллекциям, в которых реализуется неуниверсальный интерфейс IEnumerable, но не универсальный интерфейс IEnumerable<T>. В примере выше это имело бы место, если бы переменная customers относилась к типу ArrayList.

#### Выражения вырожденных запросов

Выражение запроса вида

from x in e select x

переводится в

( e ) . Select ( x => x )

Пример:

from c in customers  
select c

переводится в

customers.Select(c => c)

Выражение вырожденного запроса просто выбирает элементы из источника. На более позднем этапе перевода происходит удаление вырожденных запросов, созданных на других этапах перевода, путем их замены на соответствующие источники. Тем не менее важно обеспечить, чтобы результат выражения запроса никогда не был равен самому исходному объекту, так как это раскроет тип и идентификатор источника клиенту, направившему запрос. С этой целью на данном этапе обеспечивается защита вырожденных запросов, написанных непосредственно в исходном коде, путем явного вызова оператора Select для источника. После этого сами разработчики, реализующие Select и другие операторы запросов, должны сделать так, чтобы эти операторы никогда не возвращали сам исходный объект.

#### Предложения from, let, where, join и orderby

Выражение запроса с вторым предложением from, за которым следует предложение select,

from x1 in e1  
from x2 in e2  
select v

переводится в

( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )

Выражение запроса с вторым предложением from, за которым следует предложение, отличное от select,

from x1 in e1  
from x2 in e2  
…

переводится в

from \* in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )  
…

Выражение запроса с предложением let

from x in e  
let y = f  
…

переводится в

from \* in ( e ) . Select ( x => new { x , y = f } )  
…

Выражение запроса с предложением where

from x in e  
where f  
…

переводится в

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

Выражение запроса с предложением join без ключевого слова into, за которым следует предложение select

from x1 in e1  
join x2 in e2 on k1 equals k2  
select v

переводится в

( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )

Выражение запроса с предложением join без ключевого слова into, за которым следуем предложение, отличное от select

from x1 in e1  
join x2 in e2 on k1 equals k2   
…

переводится в

from \* in ( e1 ) . Join(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })  
…

Выражение запроса с предложением join с ключевым словом into, за которым следует предложение select

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
select v

переводится в

( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )

Выражение запроса с предложением join с ключевым словом into, за которым следуем предложение, отличное от select

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
…

переводится в

from \* in ( e1 ) . GroupJoin(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })  
…

Выражение запроса с предложением orderby

from x in e  
orderby k1 , k2 , … , kn  
…

переводится в

from x in ( e ) .   
OrderBy ( x => k1 ) .   
ThenBy ( x => k2 ) .  
 … .   
ThenBy ( x => kn )  
…

Если в предложении упорядочения указывается индикатор направления descending, вместо него вызывается метод OrderByDescending или ThenByDescending.

В следующих преобразованиях предполагается, что в каждом выражении запроса нет предложений let, where, join и orderby, и есть не больше одного начального предложения from.

Пример:

from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }

переводится в

customers.  
SelectMany(c => c.Orders,  
 (c,o) => new { c.Name, o.OrderID, o.Total }  
)

Пример:

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

переводится в

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

конечный перевод имеет вид

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 })

где x — идентификатор, созданный компилятором, который в других условиях является невидимым и недоступным.

Пример:

from o in orders  
let t = o.Details.Sum(d => d.UnitPrice \* d.Quantity)  
where t >= 1000  
select new { o.OrderID, Total = t }

переводится в

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 }

конечный перевод имеет вид

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 })

где x — идентификатор, созданный компилятором, который в других условиях является невидимым и недоступным.

Пример:

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
select new { c.Name, o.OrderDate, o.Total }

переводится в

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c.Name, o.OrderDate, o.Total })

Пример:

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 }

переводится в

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 }

конечный перевод имеет вид

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)

где x и y — идентификаторы, созданные компилятором, которые в других условиях являются невидимыми и недоступными.

Пример:

from o in orders  
orderby o.Customer.Name, o.Total descending  
select o

имеет конечный перевод

orders.  
OrderBy(o => o.Customer.Name).  
ThenByDescending(o => o.Total)

#### Предложения select

Выражение запроса вида

from x in e select v

переводится в

( e ) . Select ( x => v )

за исключением случая, когда v является идентификатором x, тогда перевод имеет вид просто

( e )

Пример

from c in customers.Where(c => c.City == “London”)  
select c

переводится просто в

customers.Where(c => c.City == “London”)

#### Предложения groupby

Выражение запроса вида

from x in e group v by k

переводится в

( e ) . GroupBy ( x => k , x => v )

за исключением случая, когда v является идентификатором x, тогда перевод имеет вид

( e ) . GroupBy ( x => k )

Пример:

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

переводится в

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

#### Прозрачные идентификаторы

При некоторых переводах вставляются переменные диапазона с прозрачными идентификаторами, обозначенными \*. Прозрачные идентификаторы не являются правильным языковым средством, они существуют только в виде промежуточного этапа в процессе перевода выражений запросов.

Когда при переводе запроса вставляется прозрачный идентификатор, дальнейшие этапы перевода распространяют прозрачный идентификатор в анонимные функции и анонимные инициализаторы объектов. В таких контекстах прозрачные идентификаторы функционируют следующим образом.

* Когда прозрачный идентификатор является параметром анонимной функции, члены связанного анонимного типа автоматически оказываются в области действия в теле анонимной функции.
* Когда член с прозрачным идентификатором находится в области действия, члены этого члена также находятся в области действия.
* Когда прозрачный идентификатор оказывается в роли декларатора члена в инициализаторе анонимного объекта, он создает член с прозрачным идентификатором.

На этапах перевода, описанных выше, прозрачные идентификаторы всегда создаются вместе с анонимными типами с намерением сохранить несколько переменных диапазона в виде членов одного объекта. В реализации C# для группирования нескольких переменных диапазона можно использовать механизм, отличный от анонимных типов. В следующих примерах перевода предполагается использование анонимных типов и показывается, как при переводе можно избавиться от прозрачных идентификаторов.

Пример:

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }

переводится в

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }

и далее переводится в

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(\* => o.Total).  
Select(\* => new { c.Name, o.Total })

что после удаления прозрачных идентификаторов эквивалентно

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.Total })

где x — идентификатор, созданный компилятором, который в других условиях является невидимым и недоступным.

Пример:

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 }

переводится в

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 }

что дальше сокращается до

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 })

конечный перевод имеет вид

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 })

где x? y и z — идентификаторы, созданные компилятором, которые в других условиях являются невидимыми и недоступными.

### Шаблон выражения запроса

Шаблон выражения запроса задает шаблон методов, которые могут реализовываться в типах для поддержки выражений запросов. Поскольку выражения запросов переводятся в вызовы методов с помощью синтаксического сопоставления, в типах можно реализовывать шаблоны выражений запроса с большой гибкостью. Например, методы шаблона можно реализовать в виде методов экземпляра или методов расширения, потому что они имеют одинаковый синтаксис вызова, а методы могут запрашивать делегаты или деревья выражений, потому что анонимные функции можно преобразовывать в оба типа.

Ниже представлен рекомендуемый формат универсального типа C<T>, который поддерживает шаблон выражения запроса. Универсальный тип используется, чтобы продемонстрировать правильные отношения между параметрами и результирующими типами, но шаблон также можно реализовать и для неуниверсальных типов.

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; }  
}

В методах выше используются универсальные типы делегатов Func<T1, R> и Func<T1, T2, R>, но в них так же успешно можно было бы использовать другие типы делегатов или деревьев выражений с такими же отношениями между типами параметров и результатов.

Обратите внимание на рекомендуемое отношение между C<T> и O<T>, которое гарантирует, что методы ThenBy и ThenByDescending будут доступны только для результата операторов OrderBy или OrderByDescending. Также обратите внимание на рекомендуемый формат результата оператора GroupBy — последовательность последовательностей, где каждая внутренняя последовательность имеет дополнительное свойство Key.

Пространство имен System.Linq предоставляет реализацию шаблона операторов запроса для любого типа, в котором реализуется интерфейс System.Collections.Generic.IEnumerable<T>.

## Операторы присваивания

Операторы присваивания назначают новое значение переменной, свойству, событию или элементу индексатора.

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

Левый операнд присваивания должен быть выражением с классом переменной, доступа к свойству, доступа к индексатору или доступа к событию.

Оператор = называется простым оператором присваивания. Он присваивает значение правого операнда переменной, свойству или элементу индексатора, который представлен левым операндом. Левый операнд оператора простого присваивания не может быть доступом к событию (за исключением случая, описанного в разделе §10.8.1). Простой оператор присваивания описывается в разделе §7.17.1.

Операторы присваивания, отличные от оператора =, называются сложными операторами присваивания. Эти операторы выполняют указанную операцию для двух операндов, а затем присваивают результирующее значение переменной, свойству или элементу индексатора, представленному левым операндом. Сложные операторы присваивания описываются в разделе §7.17.2.

Операторы += и -= с выражением доступа к событию в качестве левого операнда называются операторами присвоения события. Вместе с выражением доступа к событию в качестве левого операнда не разрешается использовать ни один другой оператор присвоения. Операторы присваивания события описываются в разделе §7.17.3.

Операторы присваивания имеют правую ассоциативность; это означает, что операции группируются справа налево. Например, выражение вида a = b = c вычисляется как a = (b = c).

### Простое присваивание

Оператор = называется простым оператором присваивания.

Если левый операнд выражения простого присвоения имеет форму E.P или E[Ei], где E имеет тип времени компиляции dynamic, то операция присвоения динамически привязывается (§7.2.2). В этом случае выражение присваивания имеет тип времени компиляции dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения на основе типа E времени выполнения.

При простом присваивании правый операнд должен быть выражением, которое может быть неявно преобразовано в тип левого операнда. Операция присваивает значение правого операнда переменной, свойству или элементу индексатора, который представлен левым операндом.

Результатом выражения простого присваивания является присваивание значения левому операнду. Результат имеет такой же тип, что и левый операнд, и всегда классифицируется как значение.

Если левый операнд представляет собой свойство или доступ к индексатору, то у свойства или индексатора должен быть метод доступа set. В противном случае возникает ошибка времени привязки.

Во время выполнения обработка простого присваивания вида x = y включает следующие этапы.

* Если x классифицируется как переменная, то:
* x вычисляется для создания переменной.
* Вычисляется y и при необходимости преобразуется в тип x с помощью неявного преобразования (§6.1).
* Если переменная, представленная x, является элементом массива ссылочного типа, то во время выполнения проводится проверка с целью убедиться, что значение, вычисленное для y совместимо с экземпляром массива, элементом которого является x. Проверка оказывается успешной, если y равно null или если существует неявное преобразование значения ссылочного типа (§6.1.6) из фактического типа экземпляра, на который ссылается y, в фактический тип элемента экземпляра массива, содержащего x. В противном случае создается исключение System.ArrayTypeMismatchException.
* Значение, получающееся после вычисления и преобразования y, сохраняется в расположении, которое задается значением x.
* Если x классифицируется как свойство или доступ к индексатору, то:
* Вычисляются выражение экземпляра (если x не имеет тип static) и список аргументов (если x является доступом к индексатору), связанные с x, и полученные результаты используются при последующем вызове метода доступа set.
* Вычисляется y и при необходимости преобразуется в тип x с помощью неявного преобразования (§6.1).
* Вызывается метод доступа set для x со значением, вычисленным для y в качестве его аргумента value.

По правилам ковариации массива (§12.5) значение массива типа A[] может быть ссылкой на экземпляр массива типа B[], если существует неявное преобразование ссылочного типа из B в A. В соответствии с этими правилами присваивание элементу массива ссылочного типа требует проведения во время выполнения проверки с целью убедиться, что присваиваемое значение совместимо с экземпляром массива. В этом примере

string[] sa = new string[10];  
object[] oa = sa;

oa[0] = null; // Ok  
oa[1] = "Hello"; // Ok  
oa[2] = new ArrayList(); // ArrayTypeMismatchException

последнее присвоение приводит к исключению System.ArrayTypeMismatchException, потому что экземпляр ArrayList нельзя сохранить в элементе типа string[].

Когда назначением присваивания является свойство или индексатор типа структуры, выражение экземпляра, связанное со свойством или доступом к индексатору, должно классифицироваться как переменная. Если выражение экземпляра классифицировано как значение, возникнет ошибка времени привязки. В силу §7.6.4 такое же правило применяется к полям.

При наличии объявлений:

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; }  
 }  
}

в примере

Point p = new Point();  
p.X = 100;  
p.Y = 100;  
Rectangle r = new Rectangle();  
r.A = new Point(10, 10);  
r.B = p;

присвоения элементам p.X, p.Y, r.A и r.B разрешены, поскольку p и r являются переменными. Однако в примере

Rectangle r = new Rectangle();  
r.A.X = 10;  
r.A.Y = 10;  
r.B.X = 100;  
r.B.Y = 100;

присвоения будут недопустимыми, поскольку r.A и r.B не являются переменными.

### Сложное присваивание

Если левый операнд выражения составного присвоения имеет форму E.P или E[Ei], где E имеет тип времени компиляции dynamic, то операция присвоения динамически привязывается (§7.2.2). В этом случае выражение присваивания имеет тип времени компиляции dynamic, а разрешение, приведенное ниже, будет иметь место во время выполнения на основе типа E времени выполнения.

Операция вида x op= y обрабатывается с применением разрешения перегрузки (§7.3.4), как если бы операция записывалась в виде x op y. Тогда

* Если тип возвращаемого значения выбранного оператора может быть неявно преобразован в тип x, то операция вычисляется как x = x op y, за исключением того, что x вычисляется только один раз.
* Если выбранный оператор является стандартным оператором, то если тип возвращаемого значения выбранного оператора может быть явно преобразован в тип x и если y может быть неявно преобразован в тип x или оператор является оператором сдвига, то операция вычисляется как x = (T)(x op y), где T имеет тип x, за исключением того, что x вычисляется только один раз.
* В противном случае сложное присваивание является недопустимым и возникает ошибка времени привязки.

Выражение «вычисляется только один раз» означает, что при вычислении x op y, результаты любого составляющего выражения в x временно сохраняются и затем используются повторно при присваивании для x. Например, в присвоении A()[B()] + = C(), где A является методом, возвращающим значение типа int[], а B и C являются методами, возвращающими значение типа int, эти методы вызываются только один раз в последовательности A, B, C.

Когда левый операнд сложного присваивания является доступом к свойству или доступом к индексатору, свойство или индексатор должны иметь и метод доступа get, и метод доступа set. В противном случае возникает ошибка времени привязки.

Второе представленное выше правило позволяет в определенных контекстах вычислять x op= y как x = (T)(x op y). Существует правило, согласно которому стандартные операторы можно использовать в качестве сложных операторов, если левый операнд имеет тип sbyte, byte, short, ushort или char. Даже если оба аргумента имеют один из этих типов, стандартные операторы дают результат типа int, как описано в разделе §7.3.6.2. Таким образом, без приведения типов присвоить результат левому операнду не удастся.

Интуитивным результатом применения правила для стандартных операторов является просто то, что операция x op= y допустима, если допустимы обе операции x op y и x = y. В этом примере

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

интуитивной причиной для каждой ошибки является то, что соответствующее простое присваивание также вызвало бы ошибку.

Это также означает, что сложные операции присваивания поддерживают операции с нулификацией. В этом примере

int? i = 0;  
i += 1; // Ok

используется оператор с нулификацией +(int?,int?).

### Присваивание событий

Если левый операнд оператора += или -= классифицируется как доступ к свойству, то выражение вычисляется следующим образом.

* Вычисляет выражение экземпляра для доступа к событию (если имеется).
* Вычисляется правый операнд оператор += или -= и при необходимости преобразуется в тип левого операнда с помощью неявного преобразования (§6.1).
* Вызывается доступ к событию со списком аргументов, состоящим из правого операнда после вычисления и при необходимости после преобразования. Если оператор равен +=, вызывается метод доступа add; если оператор равен -=, вызывается метод доступа remove.

Выражение присваивания события не порождает значения. Таким образом, выражение присваивания события допустимо только в контексте операторного выражения (§8.6).

## Выражение

Выражение является либо выражением не присваивания, либо присваиванием.

expression:   
non-assignment-expression  
assignment

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

## Константные выражения

Константное выражение — это выражение, которое можно полностью вычислить во время компиляции.

constant-expression:  
expression

Константное выражение должно быть литералом null или значением одного из следующих типов: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, object, string или любого перечисляемого типа. В константных выражениях допустимы только следующие конструкции:

* Литералы (включая литерал null).
* Ссылки на члены const типов класса и структуры.
* Ссылки на члены типов перечисления.
* Ссылки на параметры const или локальные переменные.
* Вложенные выражения в скобках, которые сами являются константными выражениями.
* Выражения приведения типа при условии, что целевой тип входит в список типов, указанных выше.
* Выражения checked и unchecked.
* Выражения значения по умолчанию
* Стандартные унарные операторы +, –, ! и ~.
* Стандартные бинарные операторы +, –, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <= и >= при условии, что каждый операнд имеет тип, указанный в списке выше.
* Условный оператор ?:.

В константных выражения допустимы следующие преобразования:

* Преобразования идентификатора
* Числовые преобразования
* Преобразования перечисления
* Преобразования константных выражений
* Явные и неявные преобразования значений ссылочного типа при условии, что источник преобразования является константным выражением, которое дает значение null.

Другие преобразования константных выражений, включая упаковку, снятие упаковки и неявное преобразование значений ссылочного типа для ненулевых значений, не допускаются. Пример:

class C {  
 const object i = 5; // error: boxing conversion not permitted  
 const object str = “hello”; // error: implicit reference conversion  
}

здесь инициализация i приводит к ошибке, потому что требуется преобразование упаковки. Инициализация str вызывает ошибку, потому что требуется неявное преобразование ссылочного типа из ненулевого значения.

Если выражение соответствует требованиям, указанным выше, оно вычисляется во время компиляции. Это так, даже если выражение является вложенным в большее выражение, которое содержит неконстантные конструкции.

При вычислении константных выражений во время компиляции используются те же правила, что и при вычислении неконстантных выражений во время выполнения за исключением того, что там, где при вычислении во время выполнения возникало бы исключение, при вычислении во время компиляции возникает ошибка времени компиляции.

Если только константное выражение не будет явно помещено в контекст unchecked, переполнения, возникающие при арифметических операциях с целыми типами и преобразованиях при вычислении выражения во время компиляции, всегда будут вызывать ошибки времени компиляции (§7.19).

Константные выражения находятся в контекстах, указанных ниже. В этих контекстах ошибка времени компиляции возникает тогда, когда выражение не удается полностью вычислить во время компиляции.

* Объявления констант (§10.4).
* Объявления членов перечислений (§14.3).
* Аргументы по умолчанию для списков формальных параметров (§10.6.1)
* Метки case в операторе switch (§8.7.2).
* Операторы goto case (§8.9.3).
* Длины измерений в выражениях создания массивов (§7.6.10.4), которые включают инициализатор.
* Атрибуты (§17).

Неявное преобразование константного выражения (§6.1.9) допускает преобразование константного выражения типа int в тип sbyte, byte, short, ushort, uint или ulong при условии, что значение константного выражения находится в пределах диапазона целевого типа.

## Логические выражения

Логическое выражение — это выражение, которое дает результат типа bool; напрямую или посредством применения оператора operator true в определенных контекстах, как указано далее.

boolean-expression:  
expression

Логическим выражением является управляющее условное выражение оператора if (§8.7.1), оператора while (§8.8.1), оператора do (§8.8.2) или оператора for (§8.8.3). Для управляющего выражения оператора ?: (§7.14) действуют те же правила, что и для логического выражения, но из-за приоритетов операторов оно классифицируется как условное выражение ИЛИ.

Логическое выражение E должно обеспечивать создание значения типа bool следующим образом.

* Если E является неявно преобразуемым к типу bool, во время выполнения применяется это неявное преобразование.
* В противном случае выполняется поиск уникальной наиболее подходящей реализации оператора true в E с помощью разрешения перегрузок унарных операторов (§7.3.3), и во время выполнения применяется найденная реализация.
* Если не удается найти такой оператор, возникает ошибка времени привязки.

Тип структуры DBBool в разделе §11.4.2 является примером типа, который реализует операторы operator true и operator false.

# Операторы

Язык C# содержит множество операторов. Большинство из них будут знакомы разработчикам, имеющим опыт программирования на языках C и C++.

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

Нетерминальное выражение внедренный оператор используется для обозначения операторов, входящих в состав других операторов языка. Использование внедренного оператора вместо обычного оператора языка позволяет обходиться без операторов объявления и помеченных операторов в этих контекстах. Пример:

void F(bool b) {  
 if (b)  
 int i = 44;  
}

результатом будет ошибка времени компиляции, поскольку для оператора if требуется использовать внедренный оператор в ветви , а не оператор языка. Если бы такой код был разрешен, переменная i стала бы объявленной, но ее нельзя было бы использовать. Следует, однако, отметить, что помещение объявления i в блок делает пример допустимым.

## Конечные точки и достижимость

У каждого оператора языка имеется конечная точка. Интуитивно говоря, конечная точка оператора — это позиция, непосредственно следующая за оператором. Правилами выполнения составных операторов (операторов языка, содержащих внедренные операторы) определяется действие, которое предпринимается, когда управление достигает конечной точки внедренного оператора. Например, когда управление достигает конечной точки оператора в блоке, оно передается следующему оператору этого блока.

Если существует возможность передачи управления оператору языка в ходе выполнения, говорят, что он является достижимым. И наоборот, если возможность выполнения оператора исключена, он называется недостижимым.

В этом примере

void F() {  
 Console.WriteLine("reachable");  
 goto Label;  
 Console.WriteLine("unreachable");  
 Label:  
 Console.WriteLine("reachable");  
}

второй вызов функции Console.WriteLine недостижим, потому что он не может быть выполнен ни при каких условиях.

Если компилятором установлен факт недостижимости оператора языка, выдается предупреждение. Недостижимость оператора не рассматривается как ошибка.

Чтобы определить, достижим ли данный оператор языка или конечная точка, компилятор выполняет анализ потока управления в соответствии с правилами достижимости, установленными для каждого оператора. В ходе анализа принимаются во внимание значения константных выражений (§7.19), контролирующих поведение операторов, но возможные значения неконстантных выражений не учитываются. Иными словами, при анализе потока управления считается, что неконстантное выражение данного типа может принимать любое значение этого типа.

В этом примере

void F() {  
 const int i = 1;  
 if (i == 2) Console.WriteLine("unreachable");  
}

логическое выражение оператора if является константным выражением, поскольку оба операнда оператора == представляют собой константы. Поскольку константное выражение вычисляется во время компиляции и его значением оказывается false, вызов Console.WriteLine считается недостижимым. Однако если переменную i сделать локальной:

void F() {  
 int i = 1;  
 if (i == 2) Console.WriteLine("reachable");  
}

то вызов Console.WriteLine станет достижимым, хотя на самом деле и не будет никогда выполняться.

Блок члена-функции всегда считается достижимым. Последовательно применяя правила достижимости каждого оператора блока, можно определить достижимость любого конкретного оператора языка.

В этом примере

void F(int x) {  
 Console.WriteLine("start");  
 if (x < 0) Console.WriteLine("negative");  
}

доступность второго вызова Console.WriteLine определяется следующим образом.

* Первый оператор-выражение Console.WriteLine доступен, поскольку доступен блок метода F.
* Конечная точка первого оператора-выражения Console.WriteLine достижима, поскольку сам оператор достижим.
* Оператор if доступен, поскольку доступна конечная точка первого оператора-выражения Console.WriteLine.
* Второй оператор-выражение Console.WriteLine доступен, поскольку значением логического выражения оператора if не является константа false.

В двух ситуациях достижимость конечной точки оператора языка означает ошибку компиляции.

* Поскольку оператор switch не позволяет перейти из одного раздела switch в следующий, достижимость конечной точки списка операторов раздела switch распознается как ошибка компиляции. Обычно такая ошибка свидетельствует об отсутствии оператора break.
* Достижимость конечной точки блока функции-члена, вычисляющего значение, распознается как ошибка компиляции. Обычно такая ошибка свидетельствует об отсутствии оператора return.

## Блоки

Блок позволяет записать несколько операторов языка в контексте, обычно допускающем использование только одного оператора.

block:  
{ statement-listopt }

Блок состоит из необязательного списка операторов (§8.2.1), заключенного в фигурные скобки. Если список операторов опущен, говорят, что блок пустой.

Блок может включать операторы объявления (§8.5). Областью видимости локальной переменной или константы, объявленной в блоке, является этот блок.

Значение имени, используемого в контексте выражения, всегда должно быть одинаковым в пределах блока (§7.6.2.1).

Блок выполняется следующим образом.

* Если блок пустой, управление передается в конечную точку блока.
* Если блок непустой, управление передается в список операторов. Если управление достигает конечной точки списка операторов, после этого управление передается в конечную точку блока.

Список операторов блока считается достижимым, если сам блок является достижимым.

Конечная точка блока достижима, если блок пустой или если конечная точка списка операторов достижима.

Блок, содержащий один или несколько операторов yield (§8.14), называется блоком итератора. Блоки итераторов используются для реализации функций-членов в виде итераторов (§10.14). В отношении блоков итераторов действуют дополнительные ограничения.

* Появление оператора return в блоке итератора приводит к ошибке во время компиляции (но при этом операторы yield return разрешены).
* Наличие небезопасного контекста в блоке итератора распознается как ошибка времени компиляции (§18.1). Блок итератора всегда определяет безопасный контекст, даже если его объявление вложено в небезопасный контекст.

### Списки операторов

Список операторов состоит из одного или нескольких последовательно записанных операторов языка. Списки операторов могут входить в блоки (§8.2) и в блоки switch (§8.7.2).

statement-list:  
statement  
statement-list statement

Выполнение списка операторов начинается с передачи управления первому оператору. Если управление достигает конечной точки оператора, после этого управление передается следующему оператору. Если управление достигает конечной точки последнего оператора, после этого управление передается в конечную точку списка операторов.

Оператор списка считается достижимым, если соблюдено по крайней мере одно из следующих условий.

* Оператор является первым в списке и сам список операторов является достижимым.
* Конечная точка предыдущего оператора достижима.
* Оператор является помеченным и его метка указывается в достижимом операторе goto.

Конечная точка списка операторов достижима, если достижима конечная точка последнего оператора языка в списке.

## Пустой оператор

Пустой оператор не выполняет никаких действий.

empty-statement:  
;

Пустой оператор используется, когда в контексте, требующем наличия оператора языка, не требуется выполнять никаких операций.

Выполнение пустого оператора сводится к передаче управления в конечную точку оператора. Таким образом, конечная точка пустого оператора достижима, если достижим сам пустой оператор.

Пустой оператор может использоваться при записи оператора while с пустым телом:

bool ProcessMessage() {...}

void ProcessMessages() {  
 while (ProcessMessage())  
 ;  
}

Кроме того, с помощью пустого оператора можно объявить метку непосредственно перед закрывающей фигурной скобкой (}) блока:

void F() {  
 ...

if (done) goto exit;  
 ...

exit: ;  
}

## Помеченные операторы

Помеченный оператор позволяет предварить оператор языка меткой. Помеченные операторы разрешается использовать в блоках, но запрещается использовать как внедренные операторы.

labeled-statement:  
identifier : statement

Помеченный оператор объявляет метку, имя которой задает идентификатор. Областью видимости метки является весь блок, в котором она объявлена, включая вложенные блоки, если они имеются. Если у двух меток с одним именем перекрывающиеся области видимости, это распознается как ошибка времени компиляции.

Метка может указываться в операторах goto (§8.9.3), находящихся в области ее видимости. Это означает, что оператор goto может передавать управление в пределах блока и за его пределы, но не внутрь блока.

Метки имеют собственную область объявления и не вступают в конфликт с другими идентификаторами. Пример:

int F(int x) {  
 if (x >= 0) goto x;  
 x = -x;  
 x: return x;  
}

является допустимым; в нем имя x используется и как параметр, и как метка.

Выполнение помеченного оператора происходит точно так же, как выполнение оператора языка, следующего за меткой.

Помимо достижимости в рамках обычного потока управления, помеченный оператор может быть достижимым в случае, если его метка указывается в достижимом операторе goto. (Имеется исключение: если оператор goto находится внутри оператора try, включающего блок finally, а помеченный оператор находится вне оператора try и при этом конечная точка блока finally недоступна, то помеченный оператор недоступен из такого оператора goto.)

## Операторы объявления

Оператор объявления объявляет локальную переменную или константу. Операторы объявления разрешается использовать в блоках, но запрещается использовать как внедренные операторы.

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

### Объявления локальных переменных

Объявление локальной переменной объявляет одну или несколько локальных переменных.

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

Тип локальной переменной в объявлении локальной переменной либо непосредственно задает тип переменных, представленных в данном объявлении, либо с помощью идентификатора var показывает, что тип неявно определяется инициализатором. За типом следует список деклараторов локальных переменных, каждый из которых представляет новую переменную. Декларатор локальной переменной состоит из идентификатора, определяющего имя переменной, за которым могут следовать лексема = и инициализатор локальной переменной, присваивающий переменной начальное значение.

В контексте объявления локальной переменной, идентификатор var выступает в качестве ключевого слова контекста (§2.4.3). Если тип локальной переменной задан как var и в области видимости тип с именем var не используется, такое объявление называется объявлением неявно введенной локальной переменной: ее тип определяется типом соответствующего выражения инициализатора. В отношении объявлений неявно введенных локальных переменных действуют следующие ограничения.

* Объявление локальной переменной не может содержать несколько деклараторов локальных переменных.
* Декларатор локальной переменной должен включать инициализатор локальной переменной.
* Инициализатор локальной переменной должен представлять собой выражение.
* Выражение инициализатора должно иметь тип, определяемый во время компиляции.
* Выражение инициализатора не может ссылаться на саму объявляемую переменную.

Далее приводятся примеры неверных объявлений неявно введенных локальных переменных:

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

Значение локальной переменной получается из выражения с использованием простого имени (§7.6.2), а изменяется значение локальной переменной путем присваивания (§7.17). Локальная переменная требует определенного присваивания (§5.3) везде, где получается ее значение.

Областью видимости локальной переменной, описанной в объявлении локальной переменной, является блок, в котором встречается объявление. Ссылка на локальную переменную в позиции текста, предшествующей декларатору локальной переменной, рассматривается как ошибка. Объявление локальной переменной или константы в области видимости другой локальной переменной с тем же именем распознается как ошибка времени компиляции.

Объявление нескольких локальных переменных эквивалентно нескольким объявлениям одиночных переменных одного и того же типа. Кроме того, использование инициализатора в объявлении локальной переменной в точности эквивалентно вставке оператора присваивания непосредственно после объявления.

Пример:

void F() {  
 int x = 1, y, z = x \* 2;  
}

в точности соответствует следующем коду:

void F() {  
 int x; x = 1;  
 int y;  
 int z; z = x \* 2;  
}

В объявлении неявно введенной локальной переменной в качестве типа объявляемой переменной принимается тип выражения, используемого для инициализации переменной. Пример:

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

Эти объявления неявно введенных локальных переменных в точности эквивалентны следующим объявлениям с явным определением типа:

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>();

### Объявления локальных констант

Объявление локальной константы объявляет одну или несколько локальных констант.

local-constant-declaration:  
const type constant-declarators

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

Тип в объявлении локальной константы задает тип констант, представленных в объявлении. За типом следует список деклараторов констант, каждый из которых представляет новую константу. Декларатор константы состоит из идентификатора, определяющего имя константы, за которым следуют лексема = и константное выражение (§7.19), задающее значение константы.

Тип и константное выражение в объявлении локальной константы подчиняются тем же правилам, что и в объявлении члена-константы (§10.4).

Значение локальной константы получается из выражения с использованием простого имени (§7.6.2).

Областью видимости локальной константы является блок, в котором встречается объявление. Ссылка на локальную константу в позиции текста, предшествующей ее декларатору константы, рассматривается как ошибка. Объявление локальной переменной или константы в области видимости другой локальной константы с тем же именем распознается как ошибка времени компиляции.

Объявление нескольких локальных констант эквивалентно нескольким объявлениям одиночных констант одного и того же типа.

## Операторы-выражения

Оператор выражение служит для вычисления данного выражения. Значение, вычисленное выражением (если оно получено), не сохраняется.

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

Не все выражения разрешается использовать как операторы языка. В частности, такие выражения, как x + y и x == 1, которые просто вычисляют значение (впоследствии отбрасываемое), недопустимы в качестве операторов языка.

В ходе выполнения оператора выражения производится вычисление содержащегося в нем выражения, после чего управление передается в конечную точку оператора выражения. Конечная точка оператора выражения считается достижимой, если достижим сам оператор выражение.

## Операторы выбора

Оператор выбора выбирает для выполнения один из нескольких возможных операторов языка на основании значения заданного выражения.

selection-statement:  
if-statement  
switch-statement

### Оператор if

Оператор if выбирает оператор языка для выполнения на основании значения логического выражения.

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

Компонент else относится к лексикографически ближайшему предшествующему компоненту if, разрешенному правилами синтаксиса. Так, оператор if в виде

if (x) if (y) F(); else G();

равнозначно

if (x) {  
 if (y) {  
 F();  
 }  
 else {  
 G();  
 }  
}

Оператор if выполняется следующим образом.

* Вычисляется логическое выражение (§7.20).
* Если результатом логического выражения является true, управление передается первому внедренному оператору. Если управление достигает конечной точки этого оператора, после этого управление передается в конечную точку оператора if.
* Если результатом логического выражения является false и в операторе имеется компонент else, управление передается второму внедренному оператору. Если управление достигает конечной точки этого оператора, после этого управление передается в конечную точку оператора if.
* Если результатом логического выражения является false и компонент else отсутствует, управление передается в конечную точку оператора if.

Первый внедренный оператор в составе оператора if считается достижимым, если оператор if достижим и значением логического выражения не является константа false.

Второй внедренный оператор оператора if (если он имеется) достижим, если оператор if достижим и значением логического выражения не является константа true.

Конечная точка оператора if достижима, если достижима конечная точка по крайней мере одного из его внедренных операторов. Кроме того, конечная точка оператора if без компонента else считается достижимой, если оператор if достижим и значением логического выражения не является константа true.

### Оператор switch

Оператор switch выбирает для выполнения список операторов, метка которого соответствует значению switch-выражения.

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 :

Оператор switch состоит из ключевого слова switch, за которым следуют выражение в скобках (так называемое switch-выражение) и затем блок switch. Блок switch состоит из произвольного (возможно, нулевого) числа разделов switch, заключенных в фигурные скобки. Каждый раздел switch состоит из одной или нескольких меток switch и следующего за ними списка операторов (§8.2.1).

Определяющий тип оператора switch устанавливается switch-выражением.

* Если типом switch-выражения является sbyte, byte, short, ushort, int, uint, long, ulong, bool, char, string или перечисляемый тип, или если это обнуляемый тип, соответствующий одному из перечисленных типов, то он считается определяющим типом оператора switch.
* В противном случае должно существовать ровно одно пользовательское неявное преобразование (§6.4) типа switch-выражения в один из следующих возможных определяющих типов: sbyte, byte, short, ushort, int, uint, long, ulong, char, string или обнуляемый тип, соответствующий одному из этих типов.
* В противном случае, если такого неявного преобразования не существует или определено несколько таких неявных преобразований, происходит ошибка времени компиляции.

Константное выражение каждого раздела case должно представлять значение, тип которого допускает неявное преобразование (§6.1) в определяющий тип оператора switch. Ошибка времени компиляции возникает, если несколько меток case в одном операторе switch задают одно и то же константное значение.

В операторе switch может быть не более одной метки default.

Оператор switch выполняется следующим образом.

* Вычисляется switch-выражение, которое преобразуется в определяющий тип.
* Если одна из констант, указанных в метке case того же оператора switch, совпадает со значением switch-выражения, управление передается списку операторов, следующему за такой меткой case.
* Если ни одна из констант, указанных в метках case того же оператора switch, не совпадает со значением switch-выражения и при этом имеется метка default, управление передается списку операторов, следующему за меткой default.
* Если ни одна из констант, указанных в метках case того же оператора switch, не совпадает со значением switch-выражения и метка default отсутствует, управление передается в конечную точку оператора switch.

Если конечная точка списка операторов в разделе switch достижима, распознается ошибка времени компиляции. Это так называемое правило «запрета последовательного выполнения». Пример:

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
default:  
 CaseOthers();  
 break;  
}

является допустимым, поскольку в нем нет разделов switch с достижимой конечной точкой. В отличие от языков C и C++, при выполнении раздела switch не допускается выход в следующий раздел switch, поэтому в примере

switch (i) {  
case 0:  
 CaseZero();  
case 1:  
 CaseZeroOrOne();  
default:  
 CaseAny();  
}

возникает ошибка времени компиляции. Если после выполнения одного раздела switch должно следовать выполнение другого раздела switch, необходимо явным образом указывать оператор goto case или goto default:

switch (i) {  
case 0:  
 CaseZero();  
 goto case 1;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
default:  
 CaseAny();  
 break;  
}

В разделе switch разрешается использовать несколько меток. Пример:

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
case 2:  
default:  
 CaseTwo();  
 break;  
}

является допустимым. Здесь правило запрета последовательного выполнения не нарушается, поскольку метки case 2: и default: входят в один раздел switch.

Правило запрета последовательного выполнения позволяет избежать распространенных ошибок в программах C и C++, вызываемых случайным пропуском оператора break. Кроме того, благодаря этому правилу разделы switch оператора switch можно расставлять в произвольном порядке — это не повлияет на поведение оператора. Например, в вышеприведенном примере можно расположить разделы switch в обратном порядке, и это не отразится на выполнении оператора:

switch (i) {  
default:  
 CaseAny();  
 break;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
case 0:  
 CaseZero();  
 goto case 1;  
}

Список операторов раздела switch обычно заканчивается оператором break, goto case или goto default, но в принципе допускается любая конструкция, исключающая достижимость конечной точки списка операторов. Например, оператор while, контролируемый логическим выражением true, никогда не позволит достичь его конечной точки. Аналогично операторы throw и return всегда передают управление в другое место, и их конечные точки также недостижимы. Поэтому следующий пример будет допустимым:

switch (i) {  
case 0:  
 while (true) F();  
case 1:  
 throw new ArgumentException();  
case 2:  
 return;  
}

Определяющий тип оператора switch может быть типом string. Пример:

void DoCommand(string command) {  
 switch (command.ToLower()) {  
 case "run":  
 DoRun();  
 break;  
 case "save":  
 DoSave();  
 break;  
 case "quit":  
 DoQuit();  
 break;  
 default:  
 InvalidCommand(command);  
 break;  
 }  
}

Подобно операторам проверки равенства строк (§7.10.7), оператор switch действует без учета регистра символов и сможет выполнить данный раздел switch только при условии, что строка switch-выражения в точности совпадает с константой метки case.

Если определяющим типом оператора switch является string, в качестве константы метки case разрешается использовать значение null.

Списки операторов в блоке switch могут включать операторы объявления (§8.5). Областью видимости локальной переменной или константы, объявленной в блоке switch, является этот блок switch.

Значение имени, используемого в контексте выражения, должно быть одинаковым в пределах блока switch (§7.6.2.1).

Список операторов в разделе switch считается достижимым, если оператор switch является достижимым и соблюдено по крайней мере одно из следующих условий.

* Значение switch-выражения не является константой.
* Значением switch-выражения является константа, совпадающая с одной из меток case в разделе switch.
* Значением switch-выражения является константа, не совпадающая ни с одной из меток case, и в разделе switch имеется метка default.
* На метку switch раздела switch ссылается доступный оператор goto case или goto default.

Конечная точка оператора switch считается достижимой, если соблюдено по крайней мере одно из следующих условий.

* Оператор switch содержит доступный оператор break, осуществляющий выход из оператора switch.
* Оператор switch является достижимым, значение switch-выражения не является константой и метка default отсутствует.
* Оператор switch является достижимым, значением switch-выражения является константа, не совпадающая ни с одной из меток case, и метка default отсутствует.

## Операторы итераций

Оператор итераций повторно выполняет один и тот же внедренный оператор.

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

### Оператор while

Оператор while выполняет внедренный оператор несколько раз (возможно, ни разу) в зависимости от соблюдения условия.

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

Оператор while выполняется следующим образом.

* Вычисляется логическое выражение (§7.20).
* Если результатом логического выражения является true, управление передается внедренному оператору. Если управление достигает конечной точки внедренного оператора (возможно, в результате выполнения оператора continue), управление передается в начало оператора while.
* Если значением логического выражения является false, управление передается в конечную точку оператора while.

В операторе while внутри внедренного оператора можно использовать оператор break (§8.9.1) для передачи управления в конечную точку оператора while (т. е. для завершения итераций внедренного оператора), а также оператор continue (§8.9.2) для передачи управления в конечную точку внедренного оператора (для выполнения еще одной итерации оператора while).

Внедренный оператор в составе оператора while считается достижимым, если оператор while достижим и значением логического выражения не является константа false.

Конечная точка оператора while считается достижимой, если соблюдено по крайней мере одно из следующих условий.

* Оператор while содержит доступный оператор break, осуществляющий выход из оператора while.
* Оператор while является достижимым и значение логического выражения отлично от константы true.

### Оператор do

Оператор do выполняет внедренный оператор один или несколько раз в зависимости от соблюдения условия.

do-statement:  
do embedded-statement while ( boolean-expression ) ;

Оператор do выполняется следующим образом.

* Управление передается внедренному оператору.
* Если управление достигает конечной точки внедренного оператора (возможно, в результате выполнения оператора continue), вычисляется логическое выражение (§7.20). Если результатом логического выражения является true, управление передается в начало оператора do. В противном случае управление передается в конечную точку оператора do.

В операторе do внутри внедренного оператора можно использовать оператор break (§8.9.1) для передачи управления в конечную точку оператора do (т. е. для завершения итераций внедренного оператора), а также оператор continue (§8.9.2) для передачи управления в конечную точку внедренного оператора.

Внедренный оператор в операторе do считается достижимым, если оператор do является достижимым.

Конечная точка оператора do считается достижимой, если соблюдено по крайней мере одно из следующих условий.

* Оператор do содержит доступный оператор break, осуществляющий выход из оператора do.
* Конечная точка внедренного оператора достижима и значение логического выражения отлично от константы true.

### Оператор for

Оператор for вычисляет последовательность инициализирующих выражений и затем, пока заданное условие соблюдается, повторно выполняет внедренный оператор и вычисляет последовательность итерационных выражений.

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

Инициализатор for (если он задан) состоит из объявления локальной переменной (§8.5.1) или из списка выражений операторов (§8.6), разделенных запятыми. Область видимости локальной переменной, объявленной инициализатором for, начинается с декларатора локальной переменной и заканчивается вместе с внедренным оператором. В область видимости включаются условие for и итератор for.

Условие for (если оно задано) должно быть логическим выражением (§7.20).

Итератор for (если он задан) состоит из списка выражений операторов (§8.6), разделенных запятыми.

Оператор for выполняется следующим образом.

* Если задан инициализатор for, инициализаторы переменных или выражения операторов выполняются в порядке их записи. Этот шаг выполняется только один раз.
* Если задано условие for, оно проверяется.
* Если условие for отсутствует или результатом его вычисления является true, управление передается внедренному оператору. Если управление достигает конечной точки внедренного оператора (возможно, в результате выполнения оператора continue), после этого последовательно вычисляются выражения итератора for (если они заданы), а затем выполняется следующая итерация, начиная с проверки условия for (см. выше).
* Если условие for задано и результатом его вычисления является false, управление передается в конечную точку оператора for.

В операторе for внутри внедренного оператора можно использовать оператор break (§8.9.1) для передачи управления в конечную точку оператора for (т. е. для завершения итераций внедренного оператора), а также оператор continue (§8.9.2) для передачи управления в конечную точку внедренного оператора (для выполнения итератора for и еще одной итерации оператора for, начиная с условия for).

Внедренный оператор в операторе for считается достижимым, если соблюдено одно из следующих условий.

* Оператор for является достижимым и условие for отсутствует.
* Оператор for является достижимым, условие for задано и его значение отлично от константы false.

Конечная точка оператора for считается достижимой, если соблюдено по крайней мере одно из следующих условий.

* Оператор for содержит доступный оператор break, осуществляющий выход из оператора for.
* Оператор for является достижимым, условие for задано и его значение отлично от константы true.

### Оператор foreach

Оператор foreach осуществляет перечисление элементов коллекции, выполняя внедренный оператор для каждого элемента коллекции.

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

Тип и идентификатор в каждом операторе foreach объявляют итерационную переменную оператора. Если тип локальной переменной задан идентификатором var, и в области видимости не существует типа именованного var, говорят, что это неявно типизированная итерационная переменная, и в качестве ее типа принимается тип элементов оператора foreach, как описано ниже. Итерационная переменная соответствует локальной переменной, доступной только на чтение, область видимости которой охватывает внедренный оператор. В ходе выполнения оператора foreach итерационная переменная представляет элемент коллекции, для которого в данный момент выполняется итерация. Если внедренный оператор пытается изменить итерационную переменную (путем присваивания или с помощью операторов ++ и ‑‑) или передать ее как параметр ref или out, возникает ошибка времени компиляции.

Далее интерфейсы IEnumerable, IEnumerator, IEnumerable<T> и IEnumerator<T> для краткости называются соответствующими типами из пространств имен System.Collections и System.Collections.Generic.

Во время компиляции оператора foreach сначала определяется тип коллекции, тип перечислителя и тип элементов для выражения. Это происходит следующим образом.

* Если типом X выражения является тип массива, производится неявное преобразование ссылочного типа из X в интерфейс IEnumerable (поскольку System.Array реализует этот интерфейс). В качестве типа коллекции принимается интерфейс IEnumerable, в качестве типа перечислителя — интерфейс IEnumerator и в качестве типа элементов — тип элементов типа массива X.
* Если тип X выражения имеет модификатор dynamic, существует неявное преобразование из выражения в интерфейс IEnumerable (§6.1.8). В качестве типа коллекции принимается интерфейс IEnumerable, в качестве типа перечислителя — интерфейс IEnumerator. Если идентификатор var представлен как тип локальной переменной, то типом элемента будет dynamic, в противном случае – object.
* В противном случае следует определить, существует ли для типа X соответствующий метод GetEnumerator.
* Выполняется поиск членов для типа X с идентификатором GetEnumerator без использования аргументов типа. Если поиск членов не дает результата, или дает неоднозначный результат, или находит член, не являющийся группой методов, следует проверить наличие перечислимого интерфейса, как описано ниже. Рекомендуется выдавать предупреждение, если результатом поиска членов является что-либо, кроме группы методов или отсутствия совпадений.
* Выполняется разрешение перегрузки с использованием результирующей группы методов и пустого списка аргументов. Если разрешение перегрузки не позволяет получить применимые методы, дает неоднозначный результат или приводит к единственному наилучшему методу, который описан как статический или не открытый, следует проверить наличие перечислимого интерфейса, как описано ниже. Рекомендуется выдавать предупреждение, если результатом разрешения перегрузки является что-либо, кроме однозначно определенного открытого метода экземпляра или отсутствия применимых методов.
* Если тип возвращаемого значения E метода GetEnumerator не является типом класса, структуры или интерфейса, возникает ошибка, и больше никакие действия не выполняются.
* Выполняется поиск членов для типа E с идентификатором Current без использования аргументов типа. Если поиск членов не дает результата, приводит к ошибке или находит что-либо, кроме открытого свойства экземпляра, разрешающего чтение, возникает ошибка, и больше никакие действия не выполняются.
* Выполняется поиск членов для типа E с идентификатором MoveNext без использования аргументов типа. Если поиск членов не дает результата, приводит к ошибке или находит что-либо, кроме группы методов, возникает ошибка, и больше никакие действия не выполняются.
* Выполняется разрешение перегрузки для группы методов с использованием пустого списка аргументов. Если разрешение перегрузки не позволяет получить применимые методы, дает неоднозначный результат или приводит к единственному наилучшему методу, который описан как статический или не открытый либо возвращает значение типа, отличного от bool, возникает ошибка, и больше никакие действия не выполняются.
* В качестве типа коллекции принимается X, в качестве типа перечислителя — E, в качестве типа элементов — тип свойства Current.
* В противном случае проверяется наличие перечислимого интерфейса.
* Если среди всех типов Ti, для которых существует неявное преобразование из X в IEnumerable<Ti>, существует уникальный тип T и этот тип T не является dynamic, а для всех остальных типов Ti имеется неявное преобразование из IEnumerable<T> в IEnumerable<Ti>, то в качестве типа коллекции принимается интерфейс IEnumerable<T>, в качестве типа перечислителя — интерфейс IEnumerator<T>, и в качестве типа элементов — тип T.
* Если таких типов T несколько, возникает ошибка, и больше никакие действия не выполняются.
* Если существует неявное преобразование типа X в интерфейс System.Collections.IEnumerable, в качестве типа коллекции принимается этот интерфейс, в качестве типа перечислителя — интерфейс System.Collections.IEnumerator и в качестве типа элементов — тип object.
* В противном случае возникает ошибка, и больше никакие действия не выполняются.

Если описанная выше процедура завершается успешно, она дает однозначно определенный результат: тип коллекции C, тип перечислителя E и тип элементов T. После этого оператор foreach, заданный в виде

foreach (V v in x) внедренный оператор

развертывается следующим образом:

{  
 E e = ((C)(x)).GetEnumerator();  
 try {  
 while (e.MoveNext()) {  
 V v = (V)(T)e.Current;  
 внедренный\_оператор  
 }  
 }  
 finally {  
 … // Ликвидация e  
 }  
}

Переменная e невидима и недоступна для выражения x, внедренного оператора и любого другого исходного кода программы. Переменная v доступна только для чтения во внедренном операторе. Если не определено явное преобразование (§6.2) типа T (типа элементов) в V (тип локальной переменной в операторе foreach), возникает ошибка, и больше никакие действия не выполняются. Если значение x равно null, во время выполнения возникает исключение System.NullReferenceException.

Для любого оператора foreach разрешается использовать реализацию, в которой он обрабатывается иначе (например, для повышения быстродействия), при условии сохранения поведения, совместимого с вышеописанным развертыванием.

Размещение переменной v внутри цикла важно для определения способа ее фиксации анонимной функцией, которая возникает во внедренном операторе.

Пример:

int[] values = { 7, 9, 13 };  
Action f = null;

foreach (var value in values)  
{  
 if (f == null) f = () => Console.WriteLine("First value: " + value);  
}

f();

Если бы переменная v была объявлена за пределами цикла, она бы совместно использовалась всеми итерациями и ее значение после завершения цикла for стало окончательным значением — 13, которое было бы выведено на печать в результате вызова f. Вместо этого, поскольку каждая итерация имеет собственную переменную v, переменная, зафиксированная функцией f в первой итерации, будет по-прежнему содержать значение 7, которое и выводится на печать. (Примечание. В более ранних версиях C# переменная v объявлялась за пределами цикла while.)

Тело блока finally строится по следующим правилам.

* Если существует неявное преобразование E в интерфейс System.IDisposable, то
* Если E является необнуляемым типом значений, предложение finally развертывается в семантический эквивалент следующего:

finally {  
 ((System.IDisposable)e).Dispose();  
}

* В противном случае предложение finally развертывается в семантический эквивалент следующего:

finally {  
 if (e != null) ((System.IDisposable)e).Dispose();  
}

Исключение составляют ситуации, когда E является типом значения, или же параметром типа, экземпляром которого оказывается тип значения; в этих случаях приведение e к System.IDisposable не сопровождается упаковкой.

* Если E является запечатанным типом, предложение finally развертывается в пустой блок:

finally {  
}

* В противном случае предложение finally развертывается следующим образом:

finally {  
 System.IDisposable d = e as System.IDisposable;  
 if (d != null) d.Dispose();  
}

Локальная переменная d невидима или недоступна для любого пользовательского кода. В частности, она не вступает в конфликты с другими переменными, чьи области видимости включают блок finally.

Оператор foreach обходит элементы массива в следующем порядке. Элементы одномерных массивов обходятся в порядке возрастания индекса, начиная с индекса 0 и заканчивая индексом Length – 1. Элементы многомерных массивов обходятся сначала по возрастанию индексов самого правого измерения, затем по возрастанию индексов следующего измерения слева и так далее до самого левого измерения.

В следующем примере выводится каждое значение двумерного массива в порядке следования элементов:

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();  
 }  
}

Вывод выглядит следующим образом:

1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

В этом примере

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

в качестве типа n неявно принимается int — тип элементов массива numbers.

## Операторы перехода

Операторы перехода осуществляют безусловную передачу управления.

jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement

Точка, в которую оператор перехода передает управление, называется его целью.

Если оператор перехода находится внутри блока, а его цель — вне этого блока, говорят, что оператор перехода производит выход из блока. Оператор перехода может передавать управление за пределы блока, но он никогда не передает управление внутрь блока.

Выполнение оператора перехода усложняется при наличии сопутствующих операторов try. В отсутствие таких операторов try оператор перехода выполняет безусловную передачу управления своей цели. При его использовании вместе с операторами try схема выполнения становится более сложной. Если оператор перехода производит выход из одного или нескольких блоков try, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех сопутствующих операторов try.

В этом примере

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");  
 }  
}

перед передачей управления в цель оператора перехода выполняются блоки finally, соответствующие двум операторам try.

Вывод выглядит следующим образом:

Before break  
Innermost finally block  
Outermost finally block  
After break

### Оператор break

Оператор break осуществляет выход из ближайшего включающего оператора switch, while, do, for или foreach.

break-statement:  
break ;

Целью оператора break является конечная точка ближайшего содержащего оператора switch, while, do, for или foreach. Если оператор break не содержится ни в одном операторе switch, while, do, for или foreach, возникает ошибка времени компиляции.

Если несколько операторов switch, while, do, for или foreach вложены друг в друга, оператор break применяется только к самому внутреннему из них. Для передачи управления с переходом через несколько уровней вложенности следует использовать оператор goto (§8.9.3).

Оператор break не дает возможности выйти из блока finally (§8.10). Если оператор break встречается внутри блока finally, цель оператора break должна находиться в том же блоке finally, в противном случае возникнет ошибка времени компиляции.

Оператор break выполняется следующим образом.

* Если оператор break выполняет выход из одного или нескольких блоков try, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех сопутствующих операторов try.
* Управление передается цели оператора break.

Поскольку оператор break осуществляет безусловную передачу управления в другое место, конечная точка оператора break никогда не будет достижима.

### Оператор continue

Оператор continue начинает новую итерацию ближайшего содержащего оператора while, do, for или foreach.

continue-statement:  
continue ;

Целью оператора continue является конечная точка внедренного оператора в составе ближайшего содержащего оператора while, do, for или foreach. Если оператор continue не содержится ни в одном операторе while, do, for или foreach, возникает ошибка времени компиляции.

Если несколько операторов while, do, for или foreach вложены друг в друга, оператор continue применяется только к самому внутреннему из них. Для передачи управления с переходом через несколько уровней вложенности следует использовать оператор goto (§8.9.3).

Оператор continue не дает возможности выйти из блока finally (§8.10). Если оператор continue встречается внутри блока finally, цель оператора continue должна находиться в том же блоке finally, в противном случае возникнет ошибка времени компиляции.

Оператор continue выполняется следующим образом.

* Если оператор continue выполняет выход из одного или нескольких блоков try, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех сопутствующих операторов try.
* Управление передается цели оператора continue.

Поскольку оператор continue осуществляет безусловную передачу управления в другое место, конечная точка оператора continue никогда не будет достижима.

### Оператор goto

Оператор goto передает управление оператору, обозначенному меткой.

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

Целью оператора goto идентификатор является помеченный оператор с данной меткой. Если метка с указанным именем не существует в текущей функции-члене или если оператор goto не находится в области видимости метки, возникает ошибка времени компиляции. Это правило позволяет использовать оператор goto для передачи управления за пределы вложенной области видимости, но не внутрь вложенной области видимости. В этом примере

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);  
 }  
 }  
}

оператор goto служит для передачи управления из вложенной области видимости.

Целью оператора goto case является список операторов в ближайшем содержащем операторе switch (§8.7.2), который содержит метку case с данным константным значением. Если оператор goto case не содержится ни в одном операторе switch, или если константное выражение не допускает неявное преобразование (§6.1) в определяющий тип ближайшего объемлющего оператора switch, или если ближайший объемлющий оператор switch не содержит метку case с данным константным значением, возникает ошибка времени компиляции.

Целью оператора goto default является список операторов в ближайшем содержащем операторе switch (§8.7.2), содержащем метку default. Если оператор goto default не содержится ни в одном операторе switch или ближайший содержащий оператор switch не содержит метку default, возникает ошибка времени компиляции.

Оператор goto не дает возможности выйти из блока finally (§8.10). Если оператор goto встречается внутри блока finally, цель оператора goto должна находиться в том же блоке finally, в противном случае возникнет ошибка времени компиляции.

Оператор goto выполняется следующим образом.

* Если оператор goto выполняет выход из одного или нескольких блоков try, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех сопутствующих операторов try.
* Управление передается цели оператора goto.

Поскольку оператор goto осуществляет безусловную передачу управления в другое место, конечная точка оператора goto никогда не будет достижима.

### Оператор return

Оператор return возвращает управление текущему объекту, вызвавшему функцию, в которой используется оператор return.

return-statement:  
return expressionopt ;

Оператор return без выражения может использоваться только в функции-члене, не вычисляющей значение, т. е. в методе с типом результата void (§10.6.10), методе доступа set для свойства или индексатора, в методах доступа add и remove для события, в конструкторе экземпляров, статическом конструкторе или деструкторе.

Оператор return с выражением может использоваться только в функции-члене, вычисляющей значение, т. е. в методе с типом результата, отличным от void, в методе доступа get для свойства или индексатора или в операторе, определенном пользователем. Должно существовать неявное преобразование (§6.1) типа выражения в тип возвращаемого значения функции-члена, содержащей оператор.

Операторы return можно также применять в выражениях анонимных функций (§7.15) и использовать при определении преобразований, существующих для таких функций.

Включение оператора return в блок finally (§8.10) вызывает ошибку времени компиляции.

Оператор return выполняется следующим образом.

* Если оператор return сопровождается выражением, это выражение вычисляется, и результат неявным способом преобразуется в тип возвращаемого значения члена-функции, содержащей оператор. Результатом преобразования становится значение, возвращенное функцией.
* Если оператор return содержится в одном или нескольких блоках try или catch, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех объемлющих операторов try.
* Если содержащая функция не является асинхронной, управление возвращается объекту, вызвавшему содержащую функцию; ему также возвращается результирующее значение, если оно существует.
* Если содержащая функция является асинхронной, управление возвращается текущему вызывающему объекту; при наличии результирующего значения оно записывается в возвращаемую задачу, как описано в разделе §10.14.1.

Поскольку оператор return осуществляет безусловную передачу управления в другое место, конечная точка оператора return никогда не будет достижима.

### Оператор throw

Оператор throw вызывает исключение.

throw-statement:  
throw expressionopt ;

Оператор throw с выражением генерирует значение, получаемое в результате вычисления выражения. Выражение должно представлять значение с типом класса System.Exception, или с типом класса, производным от System.Exception, или с типом параметра типа, чьим эффективным базовым классом является System.Exception (или его подкласс). Если результатом вычисления выражения является значение null, вместо этого создается исключение System.NullReferenceException.

Оператор throw без выражения может использоваться только в блоке catch; в этом случае оператор повторно генерирует исключение, уже обрабатываемое этим блоком catch.

Поскольку оператор throw осуществляет безусловную передачу управления в другое место, конечная точка оператора throw никогда не будет достижима.

При генерации исключения управление передается первому предложению catch в объемлющем операторе try, который может обработать исключение. Процесс, протекающий с момента генерации исключения до передачи управления в подходящий обработчик исключений, называется распространением исключения. Распространение исключения сводится к повторению нижеописанных шагов до обнаружения предложения catch, соответствующего данному исключению. В последующем описании точкой генерации первоначально считается место генерации исключения.

* В текущей функции-члене проверяется каждый оператор try, включающий точку генерации. Для каждого оператора S, начиная с самого внутреннего оператора try и до самого внешнего оператора try, выполняются следующие шаги.
* Если блок try оператора S содержит точку генерации и оператор S содержит одно или несколько предложений catch, эти предложения catch проверяются в порядке их следования в поисках подходящего обработчика для данного исключения. Первое предложение catch, в котором указан тип исключения или базовый тип для типа исключения, считается подходящим вариантом. Предложение catch общего вида (§8.10) считается подходящим для любого типа исключения. Если подходящее предложение catch найдено, распространение исключения завершается передачей управления в блок с этим предложением catch.
* В противном случае, если блок try или catch оператора S содержит точку выдачи и в операторе S имеется блок finally, управление передается в блок finally. Если блок finally генерирует еще одно исключение, обработка текущего исключения прекращается. В противном случае, когда управление достигает конечной точки блока finally, обработка текущего исключения продолжается.
* Если обработчик исключений не удается найти в ходе текущего вызова функции, этот вызов прекращается и происходит одно из следующих событий.
* Если текущая функция не является асинхронной, вышеописанные шаги повторяются для объекта, вызвавшего функцию с точкой выдачи, соответствующей оператору, из которого была вызвана функция-член.
* Если текущая функция является синхронной и возвращает задачу, исключение записывается в возвращаемую задачу, которая переводится в состояние сбоя или отмены, как описано в разделе §10.14.1.
* Если текущая функция является синхронной и возвращает значение void, отправляется уведомление контексту синхронизации текущего потока, как описано в разделе §10.14.2.
* Если обработка исключения приводит к прекращению всех вызовов функций-членов в текущем потоке, свидетельствуя о том, что в этом потоке нет обработчика для исключения, то и сам поток прекращается. Влияние такого завершения определяется реализацией.

## Оператор try

Оператор try предлагает механизм перехвата исключений, происходящих в ходе выполнения блока. Кроме того, оператор try дает возможность определить блок кода, который должен всегда выполняться при передаче управления за пределы оператора try.

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

Существует три вида операторов try:

* блок try, за которым следует один или несколько блоков catch;
* блок try, за которым следует блок finally;
* блок try, за которым следует один или несколько блоков catch и затем блок finally.

Если в предложении catch указывается тип класса, это должен быть тип System.Exception, производный от типа System.Exception или тип параметра типа с фактическим базовым классом System.Exception (или его подклассом).

Когда в предложении catch указывается и тип класса, и идентификатор, тем самым объявляется переменная исключения с данным именем и типом. Переменная исключения соответствует локальной переменной с областью видимости, охватывающей блок catch. В ходе выполнения блока catch переменная исключения представляет обрабатываемое при этом исключение. Для контроля определенного присваивания считается, что в пределах всей области видимости переменной исключения для нее обеспечено определенное присваивание.

Доступ к объекту исключения в блоке catch невозможен, если имя переменной исключения не включено в предложение catch.

Предложение catch, в котором не указан ни тип исключения, ни имя переменной исключения, называется общим предложением catch. Оператор try может содержать только одно общее предложение catch, и при его наличии оно должно быть последним предложением catch.

В некоторых языках программирования поддерживаются исключения, которые не могут быть представлены объектами, производными от System.Exception, хотя такие исключения не генерируются кодом C#. Общее предложение catch позволяет перехватывать подобные исключения. Таким образом, общее предложение catch семантически отличается от предложения с указанным типом System.Exception тем, что первое может также перехватывать исключения в программах на других языках.

В поисках обработчика для исключения производится анализ предложений catch в лексикографическом порядке. Если в предложении catch указан тот же тип, что и в одном из предыдущих предложений catch того же оператора try (или производный от него), возникает ошибка времени компиляции. Такое ограничение позволяет избежать появления недостижимых предложений catch.

Внутри блока catch с помощью оператора throw (§8.9.5) без выражения можно повторно сгенерировать исключение, перехваченное блоком catch. Присваивание значений переменной исключения не изменяет повторно генерируемого исключения.

В этом примере

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);  
 }  
 }  
}

метод F перехватывает исключение, выводит на консоль диагностическую информацию, изменяет переменную исключения и повторно генерирует исключение. Поскольку повторно генерируется исходное исключение, выводится следующее:

Exception in F: G  
Exception in Main: G

Если бы первый блок catch сгенерировал исключение e вместо повторения текущего исключения, было бы выведено следующее:

Exception in F: G  
Exception in Main: F

Для операторов break, continue и goto передача управления за пределы блока finally вызывает ошибку времени компиляции. Если оператор break, continue или goto включен в блок finally, цель оператора должна находиться в этом же блоке finally, иначе возникнет ошибка времени компиляции.

Включение оператора return в блок finally вызывает ошибку времени компиляции.

Оператор try выполняется следующим образом.

* Управление передается в блок try.
* Если управление достигает конечной точки блока try, то:
* если в операторе try имеется блок finally, выполняется блок finally;
* управление передается в конечную точку оператора try;
* Если во время выполнения блока try исключение распространяется в оператор try, происходит следующее.
* Предложения catch, если они имеются, проверяются в порядке их следования в поисках подходящего обработчика для исключения. Первое предложение catch, в котором указан тип исключения или базовый тип для типа исключения, считается подходящим вариантом. Предложение catch общего вида считается подходящим для любого типа исключения. Если подходящее предложение catch найдено, то:
* если в найденном предложении catch объявляется переменная исключения, ей присваивается объект исключения;
* управление передается в найденный блок catch;
* Если управление достигает конечной точки блока catch, то:
* если в операторе try имеется блок finally, выполняется блок finally;
* управление передается в конечную точку оператора try;
* Если во время выполнения блока catch исключение распространяется в оператор try, происходит следующее.
* если в операторе try имеется блок finally, выполняется блок finally;
* исключение распространяется в следующий объемлющий оператор try.
* Если в операторе try нет предложений catch или ни одно из предложений catch не подходит для исключения, то:
* если в операторе try имеется блок finally, выполняется блок finally;
* исключение распространяется в следующий объемлющий оператор try.

Операторы блока finally всегда выполняются после передачи управления за пределы оператора try. Это правило соблюдается независимо от того, передается ли управление в обычном потоке выполнения, в результате выполнения оператора break, continue, goto или return, или в результате распространения исключения за пределы оператора try.

Если во время выполнения блока finally генерируется исключение, которое не перехватывается в том же блоке finally, исключение распространяется в следующий объемлющий оператор try. Если в этот момент распространялось другое исключение, оно теряется. Процесс распространения исключения рассматривается далее в описании оператора throw (§8.9.5).

Блок try оператора try считается доступным, если доступен оператор try.

Блок catch оператора try считается доступным, если доступен оператор try.

Блок finally оператора try считается доступным, если доступен оператор try.

Конечная точка оператора try достижима, если выполнены следующие два условия:

* достижима конечная точка блока try или по крайней мере одного блока catch;
* доступна конечная точка блока finally при наличии блока finally.

## Операторы checked и unchecked

Операторы checked и unchecked используются для управления контекстом контроля переполнения в арифметических операциях и преобразованиях целых типов.

checked-statement:  
checked block

unchecked-statement:  
unchecked block

Оператор checked задает вычисление всех выражений в блоке в проверяемом контексте, а оператор unchecked задает вычисление всех выражений в блоке в непроверяемом контексте.

Операторы checked и unchecked в точности эквивалентны операторам выражений checked и unchecked (§7.6.12), только они применяются к блокам, а не к выражениям.

## Оператор lock

Оператор lock устанавливает взаимоисключающую блокировку для заданного объекта, выполняет оператор языка и затем снимает блокировку.

lock-statement:  
lock ( expression ) embedded-statement

Выражение оператора lock должно представлять значение, тип которого точно является ссылочным типом. Для выражения оператора lock никогда не выполняется неявное преобразование с упаковкой (§6.1.7); использование выражения для представления значения, тип которого является типом значения, вызывает ошибку времени компиляции.

Оператор lock, записанный в виде

lock (x) ...

где x — выражение ссылочного типа, в точности эквивалентен записи

bool \_\_lockWasTaken = false;  
try {  
 System.Threading.Monitor.Enter(x, ref \_\_lockWasTaken);  
 ...  
}  
finally {  
 if (\_\_lockWasTaken) System.Threading.Monitor.Exit(x);  
}

за исключением того, что x вычисляется только один раз.

При установленной взаимоисключающей блокировке код, выполняемый в том же потоке выполнения, также может установить и снять блокировку. Однако код, выполняемый в других потоках, не сможет установить блокировку, пока не будет снята данная блокировка.

Не рекомендуется блокировать объекты System.Type для синхронизации доступа к статическим данным. Этот же тип может быть заблокирован другим кодом, что приведет к взаимоблокировке. Синхронизировать доступ к статическим данных лучше путем блокировки закрытого статического объекта. Пример:

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) {  
 ...  
 }  
 }  
}

## Оператор using

Оператор using получает один или несколько ресурсов, выполняет заданный оператор языка и затем удаляет ресурсы.

using-statement:  
using ( resource-acquisition ) embedded-statement

resource-acquisition:  
local-variable-declaration  
expression

Ресурс — это класс или структура, реализующая интерфейс System.IDisposable, который состоит из одного метода без параметров с именем Dispose. Код, использующий ресурс, может вызвать метод Dispose, чтобы показать, что ресурс больше не нужен. Если не вызывать метод Dispose, ресурс будет в итоге удален автоматически в результате сборки мусора.

Если выделение ресурса задано как объявление локальной переменной, то типом объявления локальной переменной должен быть тип dynamic или тип, допускающий неявное преобразование в System.IDisposable. Если выделение ресурса задано как выражение, то это должно быть выражение типа, допускающего неявное преобразование в System.IDisposable.

Локальные переменные, объявленные при выделении ресурса, доступны только на чтение и должны включать инициализатор. Если внедренный оператор пытается изменить эти локальные переменные (путем присваивания или с помощью операторов ++ и ‑‑), получить их адрес или передать их как параметры ref или out, возникает ошибка времени компиляции.

Процесс оператора using состоит из трех частей: выделение ресурса, использование и удаление. Использование ресурса неявно включается в оператор try с предложением finally. Это предложение finally удаляет ресурс. Если выделяется ресурс null, метод Dispose не вызывается и исключение не генерируется. Если ресурс имеет тип dynamic, это означает, что он динамически преобразован путем неявного динамического преобразования (§6.1.8) к IDisposable в процессе выделения с целью обеспечения успешного преобразования перед использованием и удалением.

Оператор using, записанный в виде

using (ResourceType resource = expression) statement

может быть развернут тремя способами. Если ResourceType имеет необнуляемый тип значения, выражение имеет вид

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 ((IDisposable)resource).Dispose();  
 }  
}

Если же ResourceType является обнуляемым типом значения или ссылочным типом (за исключением dynamic), оператор развертывается следующим образом:

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 if (resource != null) ((IDisposable)resource).Dispose();  
 }  
}

В противном случае, если ResourceType является типом dynamic, оператор развертывается следующим образом:

{  
 ResourceType resource = expression;  
 IDisposable d = (IDisposable)resource;  
 try {  
 statement;  
 }  
 finally {  
 if (d != null) d.Dispose();  
 }  
}

В обоих случаях переменная resource доступна только для чтения во внедренном операторе, а переменная d недоступна и невидима для внедренного оператора.

В реализации разрешается выполнять реализацию определенного оператора using иным образом (например, для повышения быстродействия), при условии сохранения поведения, совместимого с вышеописанным развертыванием.

Оператор using, записанный в виде

using (expression) statement

может развертываться теми же тремя способами. В этом случае в качестве ResourceType неявно принимается тип выражения expression, определяемый во время компиляции, если он существует. В противном случае в качестве ResourceType используется сам интерфейс IDisposable. Переменная resource недоступна во внедренном операторе и невидима для таких операторов.

Если выделение ресурса задано в виде объявления локальной переменной, можно выделить несколько ресурсов данного типа. Оператор using, записанный в виде

using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement

в точности эквивалентен последовательности вложенных операторов using:

using (ResourceType r1 = e1)  
 using (ResourceType r2 = e2)  
 ...  
 using (ResourceType rN = eN)  
 statement

В следующем примере создается файл log.txt и в него записываются две строки текста. Затем тот же файл открывается для чтения, и содержащиеся в нем строки текста выводятся на консоль.

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);  
 }

}  
 }  
}

Поскольку классы TextWriter и TextReader реализуют интерфейс IDisposable, в примере можно использовать оператор using для того, чтобы гарантировать корректное закрытие обрабатываемого файла после операций чтения или записи.

## Оператор yield

Оператор yield используется в блоке итератора (§8.2) для выдачи значения в объект перечислителя (§10.14.4) или в перечислимый объект итератора (§10.14.5), либо для сигнализации об окончании итерации.

yield-statement:  
yield return expression ;  
yield break ;

yield не является зарезервированным словом; оно приобретает особый смысл только тогда, когда стоит непосредственно перед ключевым словом return или break. В других контекстах yield может использоваться как идентификатор.

На использование оператора yield накладывается ряд ограничений.

* Использование оператора yield (в любой из двух форм) вне тела метода, тела оператора или тела метода доступа вызывает ошибку времени компиляции.
* Использование оператора yield (в любой из двух форм) внутри анонимной функции вызывает ошибку времени компиляции.
* Использование оператора yield (в любой из двух форм) в предложении finally оператора try вызывает ошибку времени компиляции.
* Использование оператора yield return в любом месте оператора try, содержащего предложения catch, вызывает ошибку времени компиляции.

В следующем примере демонстрируется несколько допустимых и недопустимых способов использования оператора yield.

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  
}

Должно существовать неявное преобразование (§6.1) типа выражения оператора yield return в тип выдачи итератора (§10.14.3).

Оператор yield return выполняется следующим образом.

* выражение, заданное в операторе, вычисляется, неявно преобразуется к типу yield и присваивается свойству Current объекта перечислителя;
* Выполнение блока итератора приостанавливается. Если оператор yield return находится внутри одного или нескольких блоков try, соответствующие им блоки finally не выполняются в это время;
* Метод MoveNext объекта перечислителя возвращает true в вызвавшую его программу, тем самым указывая на успешный переход объекта перечислителя к следующему элементу.

Следующий вызов метода MoveNext объекта перечислителя возобновляет выполнение блока итератора с того места, где оно было приостановлено.

Оператор yield break выполняется следующим образом.

* Если оператор yield break содержится в одном или нескольких блоках try, с которыми связаны соответствующие блоки finally, управление вначале передается в блок finally самого внутреннего оператора try. Если управление достигает конечной точки блока finally, после этого управление передается в блок finally следующего объемлющего оператора try. Этот процесс повторяется до тех пор, пока не будут выполнены все блоки finally всех объемлющих операторов try.
* Управление возвращается в метод, вызвавший блок итератора. Это либо метод MoveNext, либо метод Dispose объекта перечислителя.

Поскольку оператор yield break осуществляет безусловную передачу управления в другое место, конечная точка оператора yield break никогда не будет достижима.

# Пространства имен

Программы на C# организованы с помощью пространств имен. Пространства имен используются как в качестве «внутренней» системы организации для программы, так и в качестве «внешней» системы организации — способа представления программных элементов, предоставляемых другим программам.

Директивы using (§9.4) служат для упрощения использования пространств имен.

## Единицы компиляции

Единица компиляции определяет общую структуру исходного файла. Единица компиляции состоит из 0 или более директив using, за которыми следуют 0 или более глобальных атрибутов, за которыми следуют 0 или более объявлений членов пространства имен.

compilation-unit:  
extern-alias-directivesopt using-directivesopt global-attributesopt  
 namespace-member-declarationsopt

Программа на C# состоит из одной или более единиц компиляции, каждая из которых содержится в отдельном исходном файле. При компиляции программы на C# все единицы компиляции обрабатываются совместно. Таким образом, единицы компиляции могут зависеть друг от друга, возможно циклически.

Директивы using единицы компиляции влияют на глобальные атрибуты и объявления членов пространства имен этой единицы компиляции, но не влияют на другие единицы компиляции.

Глобальные атрибуты (§17) единицы компиляции разрешают спецификацию атрибутов для конечной сборки и модуля. Сборки и модули действуют как физические контейнеры для типов. Сборка может состоять из нескольких физически отдельных модулей.

Объявления членов пространства имен каждой единицы компиляции программы размещают члены в одной области объявления, называемой глобальным пространством имен. Пример:

Файл A.cs:

class A {}

Файл B.cs:

class B {}

Эти две единицы компиляции размещаются в одно глобальное пространство имен, в данном случае объявляя два класса с полными именами A и B. Поскольку эти две единицы компиляции размещаются в одну и ту же область объявлений, было бы ошибкой, если бы в каждой из них содержалось объявление члена с одинаковым именем.

## Объявления пространства имен

Объявление пространства имен состоит из ключевого слова namespace, за которым следует имя и тело пространства имен, а затем точка с запятой (необязательно).

namespace-declaration:  
namespace qualified-identifier namespace-body ;opt

qualified-identifier:  
identifier  
qualified-identifier . identifier

namespace-body:  
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

Объявление пространства имен может быть объявлением верхнего уровня в единице компиляции или объявлением члена внутри другого объявления пространства имен. Если объявление пространства имен встречается как объявление верхнего уровня в единице компиляции, это пространство имен становится членом глобального пространства имен. Если объявление пространства имен встречается внутри другого объявления пространства имен, внутреннее пространство имен становится членом внешнего пространства имен. В обоих случаях имя пространства имен должно быть уникальным внутри содержащего пространства имен.

Пространства имен являются неявно открытыми (public); в объявление пространства имен не могут включаться модификаторы доступа.

Внутри тела пространства имен необязательные директивы using импортируют имена других пространств имен и типов, что позволяет ссылаться на них непосредственно, а не через проверенные имена. Необязательные объявления членов пространств имен размещают члены в области объявлений пространства имен. Обратите внимание, что перед любыми объявлениями членов должны стоять все директивы using.

Уточненный идентификатор объявления пространства имен может быть одиночным идентификатором или последовательностью идентификаторов, разделенных маркером «.». Последняя форма позволяет программе определять вложенные пространства имен без лексического вложения нескольких объявлений пространств имен. Например,

namespace N1.N2  
{  
 class A {}

class B {}  
}

семантически эквивалентно

namespace N1  
{  
 namespace N2  
 {  
 class A {}

class B {}  
 }  
}

Пространства имен являются открытыми, и два объявления пространств имен с одним и тем же полным именем могут размещаться в одну и ту же область объявлений (§3.3). В этом примере

namespace N1.N2  
{  
 class A {}  
}

namespace N1.N2  
{  
 class B {}  
}

два объявления пространств имен вверху размещаются в одной и той же области объявлений, объявляя в этом случае два класса с полными именами N1.N2.A и N1.N2.B. Поскольку эти два объявления размещаются в одной и той же области объявлений, было бы ошибкой, если бы в каждом содержалось объявление члена с одним и тем же именем.

## Внешние псевдонимы

Директива extern alias вводит идентификатор, служащий псевдонимом пространства имен. Спецификация пространства имен с псевдонимом является внешней по отношению к исходному коду программы и применяется также к вложенным пространствам имен пространства имен с псевдонимом.

extern-alias-directives:  
extern-alias-directive  
extern-alias-directives extern-alias-directive

extern-alias-directive:  
extern alias identifier ;

Область видимости директивы extern alias, распространяется на директивы using, глобальные атрибуты и объявления членов пространства имен незамедлительно содержащегося блока компиляции или тела пространства имен.

Внутри блока компиляции или тела пространства имен, содержащего директиву extern alias, идентификатор, введенный директивой extern alias, можно использовать для ссылки на пространство имен псевдонимов. Если в качестве идентификатора использовать слово global, произойдет ошибка времени компиляции.

Директива extern alias делает псевдоним доступным внутри отдельной единицы компиляции или тела пространства имен, но не размещает новые члены в базовой области объявлений. Иначе говоря, директива extern alias не является транзитивной, и влияет только на единицу компиляции или тело пространства имен, в котором находится.

В следующей программе объявляются и используются два внешних псевдонима, X и Y, каждый из которых представляет корень отдельной иерархии пространства имен:

extern alias X;  
extern alias Y;

class Test  
{  
 X::N.A a;  
 X::N.B b1;  
 Y::N.B b2;  
 Y::N.C c;  
}

В программе объявлено существование внешних псевдонимов X и Y, но фактические определения этих псевдонимов являются внешними для этой программы. На классы N.B с одинаковыми именами теперь можно ссылаться как X.N.B и Y.N.B, или с помощью квалификатора псевдонима пространства имен как X::N.B и Y::N.B. Если в программе объявлен внешний псевдоним, для которого не дано внешнее определение, происходит ошибка.

## Директивы using

Директивы using упрощают использование пространств имен и типов, определенных в других пространствах имен. Директивы using влияют на процесс разрешения имен пространств имен или типов (§3.8) и простых имен (§7.6.2), но, в отличие от объявлений, директивы using не размещают новые члены в базовых областях объявлений единиц компиляции или пространств имен, в которых они использованы.

using-directives:  
using-directive  
using-directives using-directive

using-directive:  
using-alias-directive  
using-namespace-directive

Директива using alias (§9.4.1) вводит псевдоним для пространства имен или типа.

Директива using namespace (§9.4.2) импортирует члены типа пространства имен.

Область директивы using распространяется на объявления членов пространства имен непосредственно содержащей ее единицы компиляции или тела пространства имен. Область видимости директивы using не включает одноуровневые директивы using. Поэтому одноуровневые директивы using не влияют друг на друга и могут быть записаны в любом порядке.

### Директивы using alias

Директива using alias вводит идентификатор, служащий псевдонимом для пространства имен или типа внутри непосредственно вмещающей единицы компиляции или тела пространства имен.

using-alias-directive:  
using identifier = namespace-or-type-name ;

Внутри объявлений членов в единице компиляции или в теле пространства имен, содержащем директиву using alias, идентификатор, введенный директивой using alias, можно использовать для ссылки на данное пространство имен или тип. Пример:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A;

class B: A {}  
}

В приведенном примере внутри объявлений членов в пространстве имен N3, A является псевдонимом для N1.N2.A, и таким образом класс N3.B является производным от класса N1.N2.A. Такой же результат можно получить, создав псевдоним R для N1.N2 и затем ссылаясь на R.A:

namespace N3  
{  
 using R = N1.N2;

class B: R.A {}  
}

Идентификатор в директиве using alias должен быть уникальным внутри области объявлений единицы компиляции или пространства имен, непосредственно содержащих директиву using alias. Пример:

namespace N3  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A; // Error, A already exists  
}

В приведенном примере N3 уже содержит член A, поэтому использование этого идентификатора в директиве using alias вызовет ошибку времени компиляции. Аналогично, произойдет ошибка времени компиляции, если две или более директивы using alias в одной и той же единице компиляции или пространстве имен объявляют псевдонимы с одним и тем же именем.

Директива using alias делает псевдоним доступным внутри отдельной единицы компиляции или тела пространства имен, но не размещает новые члены в базовой области объявлений. Иначе говоря, директива using alias не является транзитивной, и влияет только на единицу компиляции или тело пространства имен, в котором находится. В этом примере

namespace N3  
{  
 using R = N1.N2;  
}

namespace N3  
{  
 class B: R.A {} // Error, R unknown  
}

область директивы using alias, которая вводит R, распространяется только на объявления членов в теле пространства имен, в котором она содержится, так что R неизвестно в объявлении второго пространства имен. Однако если поместить директиву using alias в содержащую единицу компиляции, то этот псевдоним будет доступен внутри обоих объявлений пространств имен:

using R = N1.N2;

namespace N3  
{  
 class B: R.A {}  
}

namespace N3  
{  
 class C: R.A {}  
}

Как и регулярные члены, имена, введенные директивой using alias, скрыты членами с таким же именем во вложенных областях. В этом примере

using R = N1.N2;

namespace N3  
{  
 class R {}

class B: R.A {} // Error, R has no member A  
}

ссылка на R.A в объявлении B приводит к ошибке времени компиляции, так как R ссылается на N3.R, а не на N1.N2.

Порядок, в котором записаны директивы using alias, не имеет значения, а на разрешение имени пространства имен или типа, на которое ссылается директива using alias, не влияет ни сама директива using alias, ни другие директивы using в непосредственно содержащей единице компиляции или теле пространства имен. Другими словами, имя типа или пространства имен директивы using alias разрешается точно так же, как если бы непосредственно содержащие директиву блок компиляции или тело пространства имен не имели директив using. Однако на директиву using alias могут повлиять директивы extern alias из непосредственно содержащего блока компиляции или тела пространства имен. В этом примере

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  
}

последняя директива using alias приводит к ошибке времени компиляции, так как на нее не влияет первая директива using alias. Первая директива using alias не приводит к ошибке, поскольку область внешнего псевдонима E включает директиву using alias.

Директивой using alias можно создать псевдоним для любого пространства имен или типа, включая пространство имен, внутри которого она находится, и любое пространство имен или тип, вложенные в это пространство имен.

Доступ к пространству имен или типу через псевдоним дает точно такой же результат, как доступ к этому пространству имен или типу через его объявленное имя. Например, если дана

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  
 }  
}

имена N1.N2.A, R1.N2.A и R2.A эквивалентны и все они ссылаются на класс, полное имя которого N1.N2.A.

С помощью псевдонимов можно именовать закрытый сформированный тип, но нельзя именовать объявление несвязанного универсального типа, не предоставляя аргументы типа. Пример:

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  
}

### Директивы using namespace

Директива using namespace импортирует типы, содержащиеся в пространстве имен, в непосредственно вмещающую единицу компиляции или тело пространства имен, что позволяет использовать идентификатор каждого типа без уточнения.

using-namespace-directive:  
using namespace-name ;

Внутри объявлений членов в единице компиляции или теле пространства имен, содержащем директиву using namespace, на типы в данном пространстве имен можно ссылаться непосредственно. Пример:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1.N2;

class B: A {}  
}

В данном примере внутри объявлений членов в пространстве имен N3 члены типа N1.N2 доступны непосредственно, и таким образом класс N3.B является производным от класса N1.N2.A.

Директива using namespace импортирует типы, содержащиеся в данном пространстве имен, но не импортирует вложенные пространства имен. В этом примере

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

class B: N2.A {} // Error, N2 unknown  
}

директива using namespace импортирует типы, содержащиеся в N1, но не пространства имен, вложенные в N1. Таким образом, ссылка на N2.A в объявлении B приводит к ошибке времени компиляции, так как в области нет членов с именем N2.

В отличие от директивы using alias, директива using namespace может импортировать типы, идентификаторы которых уже определены внутри вмещающей единицы компиляции или тела пространства имен. В действительности, имена, импортированные директивой using namespace, скрыты членами с тем же именем во вмещающей единице компиляции или теле пространства имен. Пример:

namespace N1.N2  
{  
 class A {}

class B {}  
}

namespace N3  
{  
 using N1.N2;

class A {}  
}

В данном случае внутри объявлений членов в пространстве имен N3, A ссылается на N3.A, а не на N1.N2.A.

Если более одного пространства имен, импортированных директивами using namespace в одну и ту же единицу компиляции или тело пространства имен, содержат типы с одинаковым именем, ссылки на эти типы считаются неоднозначными. В этом примере

namespace N1  
{  
 class A {}  
}

namespace N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

using N2;

class B: A {} // Error, A is ambiguous  
}

N1 и N2 содержат член A, а поскольку N3 импортирует оба, ссылка на A в N3 дает ошибку времени компиляции. В этой ситуации можно разрешить конфликт либо квалифицируя ссылки на A, либо введя директиву using alias, выбирающую отдельный A. Пример:

namespace N3  
{  
 using N1;

using N2;

using A = N1.A;

class B: A {} // A means N1.A  
}

Как и директива using alias, директива using namespace не размещает новые члены в базовой области объявлений единицы компиляции или пространства имен, а влияет только на единицу компиляции или тело пространства имен, в котором находится.

Имя пространства имен, на которое ссылается директива using namespace, разрешается таким же образом, как имя пространства имен или типа, на которое ссылается директива using alias. Таким образом, директивы using namespace в одной и той же единице компиляции или теле пространства имен не влияют друг на друга и могут быть записаны в любом порядке.

## Члены пространства имен

Объявление члена пространства имен является либо объявлением пространства имен (§9.2), либо объявлением типа (§9.6).

namespace-member-declarations:  
namespace-member-declaration  
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:  
namespace-declaration  
type-declaration

В единице компиляции или в теле пространства имен могут содержаться объявления членов пространства имен, и эти объявления размещают новые члены в базовой области объявлений содержащей единицы компиляции или тела пространства имен.

## Объявления типов

Объявление типа является объявлением класса (§10.1), объявлением структуры (§11.1), объявлением интерфейса (§13.1), объявлением перечисляемого типа (§14.1) или объявлением делегата (§15.1).

type-declaration:  
class-declaration  
struct-declaration  
interface-declaration  
enum-declaration  
delegate-declaration

Объявление типа может быть объявлением верхнего уровня в единице компиляции или объявлением члена внутри пространства имен, класса или структуры.

Если объявление типа для типа T является объявлением верхнего уровня в единице компиляции, полное имя вновь объявленного типа просто T. Если объявление типа для типа T находится внутри пространства имен, класса или структуры, то полное имя вновь объявленного типа N.T, где N является полным именем содержащего пространства имен, класса или структуры.

Тип, объявленный внутри класса или структуры, называется вложенным типом (§10.3.8).

Разрешенные модификаторы доступа и доступ по умолчанию для объявления типа зависит от контекста, в котором имеет место объявление (§3.5.1):

* типы, объявленные в единицах компиляции или в пространствах имен, могут иметь доступ public или internal. По умолчанию — доступ internal.
* типы, объявленные в классах, могут иметь доступ public, protected internal, protected internal или private. По умолчанию — доступ private.
* типы, объявленные в структурах, могут иметь доступ public, internal или private. По умолчанию — доступ private.

## Квалификаторы псевдонима пространства имен

Квалификатор псевдонима пространства имен :: дает возможность гарантировать, что на поиск имени типа не будет влиять введение новых типов и членов. Квалификатор псевдонима пространства имен всегда стоит между двумя идентификаторами, которые называют левым и правым. В отличие от регулярного квалификатора ., поиск левого идентификатора квалификатора :: выполняется только как поиск псевдонима extern или using.

Уточненный член псевдонима определяется следующим образом:

qualified-alias-member:  
identifier :: identifier type-argument-listopt

Уточненный член псевдонима можно использовать как имя пространства имен или типа (§3.8) или как левый операнд доступа к члену (§7.6.4).

Уточненный член псевдонима может принимать одну из двух форм:

* N::I<A1, ..., AK>, где N и I представляют идентификаторы, а <A1, ..., AK> — список аргументов типа. (K всегда по меньшей мере единица);
* N::I, где N и I представляют идентификаторы. (В этом случае K считается равным нулю.)

При использовании этой нотации значение уточненного члена псевдонима определяется следующим образом:

* если N является идентификатором global, то поиск I ведется в глобальном пространстве имен:
* если в глобальном пространстве имен содержится пространство имен с именем I, а K равно нулю, то уточненный член псевдонима ссылается на это пространство имен;
* иначе, если в глобальном пространстве имен содержится неуниверсальный тип с именем I, а K равно нулю, то уточненный член псевдонима ссылается на этот тип;
* иначе, если в глобальном пространстве имен содержится тип с именем I, у которого имеется K параметров типа, то уточненный член псевдонима ссылается на этот тип, сформированный с данными аргументами типа;
* иначе уточненный член псевдонима не определен и выдается ошибка времени компиляции.
* иначе, начиная с объявления пространства имен (§9.2), непосредственно содержащего уточненный член псевдонима (если имеется), далее для каждого вмещающего объявления пространства имен (если имеются), и заканчивая единицей компиляции, содержащей уточненный член псевдонима, оцениваются следующие шаги, пока не будет обнаружена сущность:
* если в объявлении пространства имен или в единице компиляции содержится директива using alias, связывающая N с типом, то уточненный член псевдонима не определен и выдается ошибка времени компиляции;
* иначе, если в объявлении пространства имен или в единице компиляции содержится директива extern alias или директива using alias, связывающая N с пространством имен, то:
* если в пространстве имен, связанном с N, содержится пространство имен с именем I, а K равно нулю, то уточненный член псевдонима ссылается на это пространство имен;
* иначе, если в пространстве имен, связанном с N, содержится неуниверсальный тип с именем I, а K равно нулю, то уточненный член псевдонима ссылается на этот тип;
* иначе, если в пространстве имен, связанном с N, содержится тип с именем I, у которого имеется K параметров типа, то уточненный член псевдонима ссылается на этот тип, сформированный с данными аргументами типа;
* иначе уточненный член псевдонима не определен и выдается ошибка времени компиляции.
* иначе уточненный член псевдонима не определен и выдается ошибка времени компиляции.

Обратите внимание, что использование квалификатора псевдонима пространства имен с псевдонимом, ссылающимся на тип, вызывает ошибку времени компиляции. Кроме того, если N является идентификатором global, то поиск выполняется в глобальном пространстве имен, даже если существует псевдоним using, связывающий global с типом или пространством имен.

### Уникальность псевдонимов

В каждой единице компиляции и теле пространства имен имеется отдельная область объявлений для псевдонимов extern и using. Так, хотя имя псевдонима extern или псевдонима using должно быть уникальным внутри набора псевдонимов extern и псевдонимов using, объявленных в непосредственно содержащей единице компиляции или теле пространства имен, псевдоним может иметь то же имя, что и тип или пространство имен, если он используется только с квалификатором ::.

В этом примере

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  
 }  
}

у имени A два возможных значения в теле второго пространства имен, так как и класс A, и псевдоним A находятся в области. Поэтому использование A в уточненном имени A.Stream неоднозначно и приводит к ошибке времени компиляции. Однако использование A с квалификатором :: не является ошибкой, так как поиск A ведется только как псевдонима пространства имен.

# Классы

Класс — это структура данных, которая может содержать члены данных (константы и поля), члены функций (методы, свойства, события, индексаторы, операторы, конструкторы экземпляров, деструкторы и статические конструкторы) и вложенные типы. Типы классов поддерживают наследование — механизм, в котором производный класс может расширять и сужать базовый класс.

## Объявления классов

Объявлением класса является объявление типа (§9.6), которое объявляет новый класс.

class-declaration:  
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
 class-baseopt type-parameter-constraints-clausesopt class-body ;opt

Объявление класса состоит из необязательного набора атрибутов (§17), затем следует необязательный набор модификаторов класса (§10.1.1), необязательный модификатор partial, ключевое слово class и идентификатор, именующий класс, затем следует необязательный список параметров типа (§10.1.3), необязательная спецификация базы класса (§), необязательный набор предложений ограничений параметров типа (§10.1.5), тело класса (§10.1.6) и необязательная точка с запятой.

Объявление класса не предоставляет предложения ограничений параметров типа, если не предоставляется список параметров типа.

Объявление класса, предоставляющее список параметров типа, является объявлением универсального класса. Кроме того, любой класс, вложенный в объявление универсального класса или в объявление универсальной структуры, сам является объявлением универсального класса, так как параметры типа для содержащего типа должны быть указаны для создания сформированного типа.

### Модификаторы классов

Объявление класса может включать последовательность модификаторов класса.

class-modifiers:  
class-modifier  
class-modifiers class-modifier

class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed  
static

Появление в объявлении класса одного и того же модификатора несколько раз является ошибкой времени компилирования.

Модификатор new допускается во вложенных классах. Он определяет скрытие классом унаследованного члена с тем же именем согласно описанию в разделе §10.3.4. Появление модификатора new в объявлении класса, не являющимся объявлением вложенного класса, является ошибкой времени компилирования.

Модификаторы public, protected, internal и private управляют доступом к классу. В зависимости от контекста, в котором возникает объявление класса, некоторые из данных модификаторов могут быть запрещены (§3.5.1).

Описание модификаторов abstract, sealed и static представлено в последующих разделах.

#### Абстрактные классы

Модификатор abstract используется для указания незавершенности класса и необходимости его использования только в качестве базового класса. Ниже представлены отличия абстрактного класса от неабстрактного.

* Абстрактный класс не может быть создан непосредственно, использование оператора new в абстрактном классе является ошибкой времени компилирования. Хотя есть возможность иметь переменные и значения, имеющие типы времени компиляции, являющиеся абстрактными типами, такие переменные и значения будут обязательно иметь значение null или будут содержать ссылки на экземпляры неабстрактных классов, производных от абстрактных типов.
* Абстрактный класс может (но не должен) содержать абстрактные члены.
* Абстрактный класс не может быть запечатанным.

Если неабстрактный класс является произведным от абстрактного класса, неабстрактный класс должен включать фактические реализации всех унаследованных абстрактных членов, таким образом переопределяя данные абстрактные члены. В этом примере

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  
 }  
}

В этом примере абстрактный класс A представляет абстрактный метод F. Класс B представляет дополнительный метод G, но так как он не предоставляет реализацию F, класс B должен быть также объявлен абстрактным. Класс C переопределяет F и обеспечивает фактическую реализацию. Так как в классе C нет абстрактных членов, класс C может быть (но не должен) быть абстрактным.

#### Запечатанные классы

Модификатор sealed используется для предотвращения создания из него производных классов. Если запечатанный класс указывается в качестве базового класса другого класса, возникает ошибка времени компилирования.

Запечатанный класс не может быть также абстрактным классом.

Модификатор sealed в первую очередь используется для предотвращения непреднамеренного создания из него производных классов, но также он обеспечивает определенные оптимизации выполнения. В частности, поскольку известно, что запечатанный класс не имеет производных классов, можно преобразовать вызовы виртуальной функции-члена экземпляров запечатанного класса в невиртуальные вызовы.

#### Статические классы

Модификатор static используется для пометки класса, объявленного в качестве статического класса. Для статического класса не может быть создан экземпляр, он не может быть использован в качестве типа и может содержать только статические члены. Только статический класс может содержать объявления методов расширения (§10.6.9).

На объявление статического класса накладываются следующие ограничения.

* Статический класс не может содержать модификатор sealed или abstract. Однако следует обратить внимание, что, поскольку для статического класса не может быть создан экземпляр или производный класс, его поведение соответствует поведению запечатанного и абстрактного класса одновременно.
* Статический класс не может содержать спецификацию базы класса (§10.1.4) и не может явно указывать базовый класс или список реализованных интерфейсов. Статический класс неявно наследуется из типа object.
* Статический класс может содержать только статические члены (§10.3.7). Обратите внимание, что константны и вложенные типы классифицируются как статические члены.
* Статический класс не может содержать члены с объявленной доступностью protected или protected internal.

Нарушение данных ограничений приводит к ошибке времени компилирования.

Статический класс не имеет конструкторов экземпляров. Невозможно объявить конструктор экземпляров в статическом классе; конструктор экземпляров по умолчанию (§10.11.4) не предоставляется для статического класса.

Члены статического класса автоматически не являются статическими, а объявления членов должны явно включать модификатор static (исключая константы и вложенные типы). Если класс является вложенным внутри статического внешнего класса, вложенный класс не является статическим классом, если он явно не включает модификатор static.

##### Ссылки на типы статического класса

Ссылка имени пространства имен или типа на статический класс (§3.8) допускается в следующих случаях.

* Имя пространства имен или типа T в имени пространства имен или типа формы T.I или
* Имя пространства имен или типа равно T в выражении typeof (§7.5.11) в виде typeof(T).

Ссылка первичного выражения на статический класс (§7.5) допускается в следующих случаях.

* Первичное выражение равно E в доступе к членам (§7.5.4) в форме E.I.

В любом другом контексте ссылка на статический класс является ошибкой времени компилирования. Например, использование статического класса в качестве базового класса, составного типа (§10.3.8) члена, аргумента универсального типа или ограничения параметра типа является ошибкой. Аналогичным образом статический класс не может использоваться в типе массива, типе указателя, выражении new, выражении приведения, выражении is, выражении as, выражении sizeof или в выражении значения по умолчанию.

### Модификатор partial

Модификатор partial используется для указания того, что данное объявление класса является объявлением разделяемого типа. Несколько объявлений разделяемого типа с одинаковым именем в рамках заключающего объявления пространства имен или типа формируют объявление одного типа, согласно правилам в разделе §10.2.

Распределение объявления класса в отдельных сегментах текста программы может быть полезным при создании и поддержке этих сегментов в различных контекстах. Например, одна часть объявления класса может быть создана системой, а другая часть – создана вручную. Текстовое разделение двух частей позволяет предотвратить конфликт обновлений разных частей.

### Параметры типа

Параметром типа является простой идентификатор, обозначающий заполнитель для аргумента типа, предоставленного для создания сформированного типа. Параметром типа является формальный заполнитель для типа, предоставляемого позже. И наоборот, аргументом типа (§4.4.1) является фактический тип, заменяемый для параметра типа при создании сформированного типа.

type-parameter-list:  
< type-parameters >

type-parameters:  
attributesopt type-parameter  
type-parameters , attributesopt type-parameter

type-parameter:  
identifier

Каждый параметр типа в объявлении класса определяет имя в области объявления (§3.3) данного класса. Поэтому невозможно задать аналогичное имя для другого параметра типа или члена, объявленного в данном классе. Параметр типа не может иметь одинаковое с самим типом имя.

### Спецификация базы класса

Объявление класса может содержать спецификацию базы класса, определяющую прямой базовый класс для класса и интерфейсов (§13), непосредственно реализованных классом.

class-base:  
: class-type  
: interface-type-list  
: class-type , interface-type-list

interface-type-list:  
interface-type  
interface-type-list , interface-type

Базовый класс, указанный в объявлении класса, может являться типом сформированного класса (§4.4). Базовый класс не может быть параметром типа сам по себе, хотя он может вовлекать параметры типа в области.

class Extend<V>: V {} // Error, type parameter used as base class

#### Базовые классы

Если тип класса включен в базу класса, он определяет прямой базовый класс объявленного класса. Если объявление класса не имеет базы классы или база класса перечисляет только типы интерфейса, прямым базовым классом считается object. Класс наследует члены из прямого базового класса (см. §10.3.3).

В этом примере

class A {}

class B: A {}

В этом примере класс A считается прямым базовым классом для класса B, а класс B считается производным из класса A. Так как класс A не указывает явно прямой базовый класс, его неявным прямым базовым классом является object.

Для сформированного типа класса, если базовый класс указан в объявлении универсального класса, базовый класс сформированного типа получается путем замещения каждого параметра типа в объявлении базового класса соответствующим аргументом типа сформированного типа. При объявлении универсального класса

class B<U,V> {...}

class G<T>: B<string,T[]> {...}

базовый класс сформированного типа G<int> должен быть B<string,int[]>.

Прямой базовый класс типа класса должен быть не менее доступен, чем сам тип класса (§3.5.2). Например, если класс public произведен из класса private или internal, возникает ошибка времени компилирования.

Прямой базовый класс типа класса не должен быть одним из следующих типов: System.Array, System.Delegate, System.MulticastDelegate, System.Enum или System.ValueType. Кроме того, объявление универсального класса не может использовать System.Attribute в качестве прямого или непрямого базового класса.

При определении значения спецификации прямого базового класса A для класса B прямым базовым классом B временно считается object. Это означает, что значение спецификации базового класса не может рекурсивно зависеть от самого себя. Пример:

class A<T> {

public class B{}

}

class C : A<C.B> {}

является ошибочным, поскольку в спецификации базового класса A<C.B>прямым базовым классом для C считается object, а следовательно (согласно правилам §3.8) считается, что C не содержит член B.

Базовыми классами типа класса являются прямой базовый класс и его базовые классы. Другими словами, набор базовых классов является транзитивным замыканием отношения прямого базового класса. Ссылаясь на пример выше, базовыми классами для класса B являются A и object. В этом примере

class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}

базовыми классами D<int> являются классы C<int[]>, B<IComparable<int[]>>, A и object.

За исключением класса object, каждый тип класса имеет строго один прямой базовый класс. Класс object не имеет прямого базового класса и является окончательным базовым классом для всех других классов.

Если класс B производится из класса A, возникает ошибка времени компилирования, так как A зависит от B. Класс непосредственно зависит от его прямого базового класса (при его наличии) и от класса, в который он непосредственно вложен (при его наличии). С учетом этого определения, полным набором классов, от которых зависит класс, является рефлексивное и транзитивное замыкание отношения непосредственной зависимости.

Пример:

class A: A {}

Этот пример является ошибочным, поскольку класс зависит сам от себя. Аналогично, в примере

class A: B {}

class B: C {}

class C: A {}

имеется ошибка, поскольку классы циклически зависят сами от себя. Наконец, в примере

class A: B.C {}

class B: A  
{  
 public class C {}  
}

имеется ошибка времени компилирования, так как класс A зависит от класса B.C (его прямой базовый класс), который зависит от класса B (его прямой заключающий класс), который зависит от класса A, создавая циклическую зависимость.

Обратите внимание, что класс не зависит от классов, вложенных в него. В этом примере

class A  
{  
 class B: A {}  
}

B зависит от A (так как A является его прямым базовым классом и прямым заключающим классом), но A не зависит от B (так как B не является базовым классом и заключающим классом A). Поэтому пример является допустимым.

Невозможно выполнить создание производного класса от запечатанного класса sealed. В этом примере

sealed class A {}

class B: A {} // Error, cannot derive from a sealed class

класс B является ошибочным, так как он пытается наследовать от класса A с модификатором sealed.

#### Реализация интерфейсов

Спецификация базы класса может включать список типов интерфейса, при этом считается, что класс непосредственно реализует заданные типы интерфейса. Реализация интерфейсов рассматривается более подробно в §13.4.

### Ограничения параметров типа

Объявления универсального типа и метода могут дополнительно указывать ограничения параметра типа, включая предложения ограничений параметров типа.

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 ( )

Каждое предложение ограничений параметров типа состоит из маркера where, следующих затем имени параметра типа, двоеточия и списка ограничений для данного параметра типа. Может существовать не более одного предложения where для каждого параметра типа, порядок перечисления предложений where не имеет существенного значения. Аналогично токенам get и set в методе доступа к свойству, токен where не является ключевым словом.

Список ограничений в предложении where может включать любые из представленных ниже компонентов в следующем порядке: одно первичное ограничение, одно или несколько вторичных ограничений и ограничение конструктора new().

Первичное ограничение может быть типом класса или ограничением ссылочного типа class или ограничением типа значения struct. Вторичное ограничение может быть параметром типа или типом интерфейса.

Ограничение ссылочного типа указывает, что аргумент типа, используемый для параметра типа, должен быть типом ссылки. Все типы классов, интерфейсов, делегатов, массивов и параметры типов считаются ссылочными типами (см. ниже), удовлетворяющих данному ограничению.

Ограничение типа значения указывает, что аргумент типа, используемый для параметра типа, не должен быть типом, который может иметь значение null. Все типы структур, которые не могут иметь значение null, типы перечислений и параметры типа с ограничением типа значения удовлетворяют данному ограничению. Обратите внимание, что, несмотря на классификацию в качестве типа значения, тип, который может иметь значение NULL (§4.1.10), не удовлетворяет ограничению типа значения. Параметр типа с ограничением типа значения также не может иметь ограничение конструктора.

Типы указателей никогда не могут быть аргументами типа и не удовлетворяют ограничениям ссылочного типа или типа значения.

Если ограничение является типом класса, типом интерфейса или параметром типа, данный тип указывает минимальный «базовый тип», который должен поддерживать каждый аргумент типа для данного параметра типа. При использовании сформированного типа или универсального метода аргумент типа проверяется на соответствие ограничениям параметра типа во время компиляции. Предоставленный аргумент типа должен удовлетворять условиям, определенным в разделе 4.4.4.

Ограничение типа класса должно удовлетворять следующим правилам.

* Типом должен быть тип класса.
* Тип не должен иметь модификатор sealed.
* Тип должен быть одним из следующих типов: System.Array, System.Delegate System.Enum или System.ValueType.
* Тип не должен совпадать с object. Так как все типы являются производными от object, отсутствие данного ограничения не оказывает никакого влияния.
* Максимум одно ограничение для данного параметра типа может являться типом класса.

Тип, заданный в качестве ограничения типа интерфейса, должен удовлетворять следующим правилам:

* Он должен быть типом интерфейса.
* Тип не должен быть указан более одного раза в данном предложении where.

В любом случае ограничение может включать любые параметры типа связанных объявлений типа или метода в качестве части сформированного типа, а также может включать объявляемый тип.

Любой тип класса или интерфейса, указанный в качестве ограничения параметра типа, должен быть не менее доступен (§3.5.4), чем объявляемый универсальный тип или метод.

Тип, заданный в качестве ограничения параметра типа, должен удовлетворять следующим правилам.

* Он должен быть параметром типа.
* Тип не должен быть указан более одного раза в данном предложении where.

Кроме того, не допускается наличие циклов в диаграмме зависимости параметров типа, где зависимостью является транзитивное отношение, заданное следующим.

* Если параметр типа T используется в качестве ограничения для параметра типа S, то S зависит от T.
* Если параметр типа S зависит от параметра типа T и T зависит от параметра типа U, то S зависит от U.

Согласно данному отношению, если параметр типа зависит от самого себя (прямым или косвенным образом), это является ошибкой времени компилирования.

Любые ограничения должны быть согласованы среди зависимых параметров типа. Если параметр типа S зависит от параметра типа T, выполняется следующее.

* T не должен иметь ограничение типа значения. В противном случае T эффективно запечатывается таким образом, что S будет принудительно того же типа, что и T, устраняя тем самым потребность в двух параметрах типа.
* Если S имеет ограничение типа значения, то T не должен иметь ограничения типа класса.
* Если S имеет ограничение типа класса A и T имеет ограничение типа класса B, то требуется преобразование идентификации или неявные преобразования ссылочных типов из A в B (или из B в A).
* Если S также зависит от параметра типа U, U имеет ограничение типа класса A и T имеет ограничение типа класса B, то требуется преобразование идентификации или неявные преобразования ссылочных типов из A в B (или из B в A).

S может иметь ограничение типа значения, и T может иметь ограничение ссылочного типа. Фактически это позволяет ограничить T типами System.Object, System.ValueType, System.Enum и любыми типами интерфейса.

Если предложение where для параметра типа включает ограничение конструктора (с формой new()), можно использовать оператор new для создания экземпляров типа (§7.6.10.1). Любой аргумент типа, используемый для параметра типа с ограничением конструктора, должен иметь открытый конструктор без параметров (данный конструктор неявно существует для любого типа значения) или должен быть параметром типа с ограничением типа значения или ограничением конструктора (подробные сведения см. в разделе §10.1.5).

Ниже представлены примеры ограничений.

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()  
{  
 ...  
}

Следующий пример содержит ошибку, так как он вызывает цикличность в графе зависимостей параметров типа.

class Circular<S,T>  
 where S: T  
 where T: S // Error, circularity in dependency graph  
{  
 ...  
}

Следующие примеры иллюстрируют некоторые недопустимые ситуации.

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  
{  
 ...  
}

Эффективный базовый класс параметра типа T определяется следующим образом.

* Если T не имеет первичных ограничений или ограничений параметра типа, его эффективным базовым классом является object.
* Если T имеет ограничение типа значения, его фактическим базовым классом является System.ValueType.
* Если T имеет ограничение типа класса C, но не имеет ограничений параметра типа, его эффективным базовым классом является C.
* Если T не имеет ограничения типа класса, но имеет одно или более ограничений параметра типа, его эффективным базовым классом является наивысший заключенный тип (§6.4.2) в наборе эффективных базовых классов его ограничений параметра типа. Правила соответствия обеспечивают существование наивысшего заключенного типа.
* Если T имеет ограничение типа класса и одно или более ограничений параметра типа, его эффективным базовым классом является наивысший заключенный тип (§6.4.2) в наборе, состоящем из ограничения типа класса T и эффективных базовых классов его ограничений параметра типа. Правила соответствия обеспечивают существование наивысшего заключенного типа.
* Если T имеет ограничение ссылочного типа, но не имеет ограничений типа класса, его эффективным базовым классом является object.

Для выполнения этих правил, если T имеет ограничение V, имеющее тип значения, используйте более конкретный базовый тип V типа класса. Это не может произойти в явно предоставленном ограничении, но может случиться, когда ограничения универсального метода неявным образом наследуются объявлением метода переопределения или явной реализацией метода интерфейса.

Эти правила обеспечивают то, что эффективным базовым классом всегда является тип класса.

Эффективный набор интерфейса параметра типа T определяется следующим образом.

* Если T не имеет вторичных ограничений, его эффективный набор интерфейсов пуст.
* Если T имеет ограничения типа интерфейса, но не имеет ограничений параметра типа, его эффективным набором интерфейса является его набор ограничений типа интерфейса.
* Если T не имеет ограничений типа интерфейса, но имеет ограничения параметра типа, его эффективным набор интерфейса является объединение эффективных наборов интерфейса его ограничений параметра типа.
* Если T имеет сразу ограничения типа интерфейса и ограничения параметра типа, его эффективным набор интерфейса является объединение ограничений типа интерфейса и эффективных наборов интерфейса его ограничений параметра типа.

Параметр типа считается ссылочным типом, если он имеет ограничение ссылочного типа, или если его эффективным базовым классом не является object или System.ValueType.

Значения ограниченного типа для типа параметра могут использоваться для доступа к членам экземпляров, подразумеваемых ограничениями. В этом примере

interface IPrintable  
{  
 void Print();  
}

class Printer<T> where T: IPrintable  
{  
 void PrintOne(T x) {  
 x.Print();  
 }  
}

методы интерфейса IPrintable могут быть вызваны непосредственно для x, так как, в силу ограничения, T обязан постоянно реализовывать IPrintable.

### Тело класса

Тело класса для класса определяет члены этого класса.

class-body:  
{ class-member-declarationsopt }

## Разделяемые типы

Объявление типа может быть разделено на несколько объявлений разделяемого типа. Объявление типа формируется из его частей в соответствии с правилами данного раздела, после чего оно считается единым объявлением на протяжении оставшегося времени компилирования и выполнения программы.

Объявление класса, структуры или интерфейса представляет объявление разделяемого типа, если оно включает модификатор partial. Модификатор partial не является ключевым словом и действует в качестве модификатора, только если он расположен непосредственно перед одним из ключевых слов class, struct или interface в объявлении типа или перед типом void в объявлении метода. В других контекстах его можно использовать в качестве обычного идентификатора.

Каждая часть объявления разделяемого типа должна содержать модификатор partial. Он должен иметь такое же имя и должен быть объявлен в том же объявлении пространства имен или типа, что и другие части. Модификатор partial указывает, что могут существовать дополнительные части объявления типа, но их существование не является обязательным; тип с одним объявлением может содержать модификатор partial.

Все части разделяемого типа должны быть скомпилированы вместе, чтобы они могли быть объединены во время компилирования в одно объявление типа. Разделяемые типы специально не допускают расширение уже скомпилированных типов.

Вложенные типы могут быть объявлены в нескольких частях путем использования модификатора partial. Обычно тип-контейнер объявляется также посредством partial, а каждая часть вложенного типа объявляется в другой части типа-контейнера.

Модификатор partial не допускается в объявлениях делегата или перечисляемого типа.

### Атрибуты

Атрибуты разделяемого типа определяются путем комбинирования в неопределенном порядке атрибутов каждой части. Если атрибут размещен в нескольких частях, он эквивалентен многократному указанию атрибута для типа. Например, две части:

[Attr1, Attr2("hello")]  
partial class A {}

[Attr3, Attr2("goodbye")]  
partial class A {}

эквиваленты такому объявлению:

[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]  
class A {}

Атрибуты параметров типа комбинируются аналогичным образом.

### Модификаторы

Если объявление разделяемого типа включает спецификацию доступности (модификаторы public, protected, internal и private), оно должно быть согласовано со всеми другими частями, включающими спецификацию доступности. Если ни одна из частей разделяемого типа не содержит спецификацию доступности, типу задается соответствующая доступность по умолчанию (§3.5.1).

Если одно или несколько разделяемых объявлений вложенного типа включают модификатор new, и если вложенный тип скрывает унаследованный член (§3.7.1.2), предупреждение не дается.

Если одно или несколько разделяемых объявлений класса включают модификатор abstract, класс считается абстрактным (§10.1.1.1). В противном случае класс считается неабстрактным.

Если одно или несколько разделяемых объявлений класса включают модификатор sealed, класс считается запечатанным (§10.1.1.2). В противном случае класс считается незапечатанным.

Обратите внимание, что класс не может быть одновременно абстрактным и запечатанным.

При использовании модификатора unsafe в объявлении разделяемого типа только соответствующая часть считается небезопасным контекстом (§18.1).

### Параметры и ограничения типа

Если универсальный тип объявлен в нескольких частях, каждая часть должна формулировать параметры типа. Все части должны иметь одинаковое количество параметров типа и одинаковые имена для каждого параметра типа (по порядку).

Если объявление разделяемого универсального типа содержит ограничения (предложения where), ограничения должны быть согласованы со всеми другими частями, включающими ограничения. В частности, каждая часть, содержащая ограничения, должна иметь ограничения для того же набора параметров типа, а для каждого параметра типа наборы первичных, вторичных ограничений и ограничений конструктора должны быть эквивалентны. Два набора ограничений эквивалентны, если они содержат одинаковые члены. Если ни одна часть разделяемого универсального типа не указывает ограничения параметра типа, параметры типа считаются неограниченными.

Пример:

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>  
{  
 ...  
}

Этот пример является верным, так как части, включающие ограничения (первые две), фактически задают один и тот же набор первичных, вторичных ограничений и ограничений конструктора для одного и того же набора параметров типа, соответственно.

### Базовый класс

Если объявление разделяемого класса содержит спецификацию базового класса, требуется согласование со всеми остальными частями, включающими спецификацию базового класса. Если ни одна часть разделяемого класса не включает спецификацию базового класса, базовым классом становится System.Object (§10.1.4.1).

### Базовые интерфейсы

Набором базовых интерфейсов для типа, объявленного в нескольких частях, является объединение базовых интерфейсов, указанных в каждой части. Определенный базовый интерфейс может быть именован только один раз в каждой части, но при этом нескольким частям разрешается именовать соответствующие базовые интерфейсы. Может быть только одна реализация членов любого заданного базового интерфейса.

В этом примере

partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}

набором базовых интерфейсов для класса C является IA, IB и IC.

Обычно каждая часть предоставляет реализацию интерфейсов, объявленных в данной части, однако это необязательно. Одна часть может предоставлять реализацию для интерфейса, объявленного в другой части.

partial class X  
{  
 int IComparable.CompareTo(object o) {...}  
}

partial class X: IComparable  
{  
 ...  
}

### Члены

Исключая разделяемые методы (§10.2.7), набором членов типа, объявленного в нескольких частях, является просто объединение членов, объявленных в каждой части. Тела всех частей объявления типа совместно используют одну область объявления (§3.3), а область каждого члена (§3.7) расширяется на тела всех частей. Домен доступности любого члена всегда содержит все части вмещающего типа; член private, объявленный в одной части, свободно доступен из другой части. Объявление одного члена в нескольких частях типа приводит к ошибке времени компиляции, если этот член не является типом с модификатором partial.

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;  
 }  
}

Порядок членов в типе редко имеет значение в коде C#, однако он может стать важным при взаимодействии с другими языками и средами. В таких случаях порядок членов в рамках типа, объявленного в нескольких частях, не определен.

### Разделяемые методы

Разделяемые методы могут быть определены в одной части объявления типа и реализованы в другой. Реализация является необязательной. Если ни одна из частей не реализует разделяемый метод, объявление разделяемого метода и все вызовы к нему удаляются из объявления типа, полученного путем комбинирования частей.

Разделяемые методы не могут определять модификаторы доступа, но неявно являются private. Типом возвращаемого значения разделяемого метода должен быть void, а параметры не могут иметь модификатора out. Идентификатор partial определяется в качестве специального ключевого слова в объявлении метода, только если он появляется непосредственно перед типом void, в противном случае он может использоваться в качестве обычного идентификатора. Разделяемый метод не может явно реализовывать методы интерфейсов.

Существует два типа объявлений разделяемых методов. Если телом объявления метода является точка с запятой, объявление считается определяющим объявлением разделяемого типа. Если тело представлено в качестве блока, объявление считается реализующим объявлением разделяемого метода. Среди частей объявления типа может быть только одно определяющее объявление разделяемого метода с заданной сигнатурой, а также может быть только одно реализующее объявление разделяемого метода с заданной сигнатурой. Если задано реализующее объявление разделяемого метода, должно существовать соответствующее определяющее объявление разделяемого метода, а объявления должны соответствовать следующему.

* Объявления должны иметь одинаковые модификаторы (необязательно в одинаковом порядке), имя метода, количество параметров типа и количество параметров.
* Соответствующие параметры в объявлениях должны иметь одинаковые модификаторы (необязательно в одном порядке) и одинаковые типы (отличия по модулю в именах параметров типа).
* Соответствующие параметры типа в объявлениях должны иметь одинаковые ограничения (отличия по модулю в именах параметров типа).

Реализующее объявление разделяемого метода может содержаться в той же части, что и соответствующее определяющее объявление разделяемого метода.

Только определяющий разделяемый метод участвует в разрешении перегрузки. Таким образом, независимо от наличия реализующего объявления, выражения вызова могут обращаться к вызовам разделяемого метода. Поскольку разделяемый метод всегда возвращает значение void, такие выражения вызова всегда будут являться операторами выражения. Кроме того, так как разделяемый метод является неявно private, такие операторы всегда будут возникать в той части объявления типа, в рамках которой объявлен разделяемый метод.

Если ни одна из частей объявления разделяемого типа не содержит реализующее объявление для заданного разделяемого метода, любой оператор выражения, вызывающий ее, просто удаляется из комбинированного объявления типа. Таким образом, выражение вызова, включая любые групповые выражения, не оказывает воздействия во время выполнения. Разделяемый метод сам по себе также удаляется и не будет членом скомбинированного объявления типа.

При наличии для заданного разделяемого метода реализующего объявления вызовы разделяемых методов сохраняются. Разделяемый метод инициирует объявление метода аналогично реализующему объявлению разделяемого метода, исключая следующее.

* Модификатор partial не включается.
* Атрибутами в результирующем объявлении метода являются скомбинированные атрибуты определяющего и реализующего объявлений разделяемого метода в неопределенном порядке. Дубликаты не удаляются.
* Атрибутам параметров в результирующем объявлении метода являются скомбинированные атрибуты соответствующих параметров определяющего и реализующего объявлений разделяемого метода в неопределенном порядке. Дубликаты не удаляются.

Если для разделяемого метода M задано определяющее объявление, а реализующее объявление не задано, применяются следующие ограничения.

* Создание делегата метода является ошибкой времени компилирования (§7.6.10.5).
* Ссылка на M внутри анонимной функции, преобразованной в тип дерева выражений, является ошибкой времени компилирования (§6.5.2).
* Выражения, возникающие в качестве части вызова M, не воздействуют на состояние определенного присваивания (§5.3), которое может привести к ошибкам времени компилирования.
* Метод M не может являться точкой входа приложения (§3.1).

Разделяемые методы полезны, если одна часть объявления типа должна управлять поведением другой части (например, сгенерированной программным средством). Обратите внимание на следующее объявление разделяемого класса:

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();  
}

Если данный класс скомпилирован без каких-либо других частей, определяющие объявления разделяемого метода и их вызовы будут удалены, а результирующее комбинированное объявление класса будет эквивалентно следующему.

class Customer  
{  
 string name;

public string Name {

get { return name; }

set { name = value; }  
 }  
}

Предположим, что задана другая часть, предоставляющая реализующие объявления разделяемых методов:

partial class Customer  
{  
 partial void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

partial void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

Тогда результирующее объявление скомбинированного класса будет эквивалентно следующему:

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);  
 }  
}

### Привязка имен

Несмотря на то, что каждая часть расширяемого типа должна быть объявлена в одном пространстве имен, части обычно прописываются в рамках различных объявлений пространств имен. Поэтому для каждой части могут присутствовать различные директивы using (§9.4). При интерпретации простых имен (§7.5.2) в рамках одной части учитываются только директивы using объявлений пространства имен, заключающие данную часть. Это может привести к наличию одинаковых идентификаторов с различными значениями в различных частях.

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  
 }  
}

## Члены класса

Члены класса состоят из членов, представленных его объявлениями членов класса и членами, унаследованными от прямого базового класса.

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

Члены типа класса разделены на следующие категории.

* Константы, представляющие постоянные значения, связанные с классом (§10.4).
* Поля, являющиеся переменными класса (§10.5).
* Методы, реализующие вычисления и действия, которые могут быть выполнены классом (§10.6).
* Свойства, определяющие именованные характеристики и действия, связанные с чтением и записью данных характеристик (§10.7).
* События, определяющие уведомления, которые могут быть сгенерированы классом (§10.8).
* Индексаторы, которые обеспечивают индексацию экземпляров класса аналогично (синтаксически) индексации массивов (§10.9).
* Операторы, определяющие операторы выражений, которые могут быть применены к экземплярам класса (§10.10).
* Конструкторы экземпляров, реализующие действия, требуемые для инициализации экземпляров класса (§10.11)
* Деструкторы, реализующие действия, выполняемые до отмены экземпляров класса без возможности восстановления (§10.13).
* Статические конструкторы, реализующие действия, требуемые для инициализации самого класса (§10.12)
* Типы, представляющие локальные типы класса (§10.3.8).

Все члены, которые могут содержать исполняемый код, считаются членами функций типа класса. Членами функций типа класса являются методы, свойства, события, индексаторы, операторы, конструкторы экземпляров, деструкторы и статические конструкторы данного типа класса.

Объявление класса создает новую область объявления (§3.3), а объявления члена класса, непосредственно содержащиеся в объявлении класса, представляют новые члены в данной области объявления. К объявлениям члена класса применяются следующие правила.

* Конструкторы экземпляров, деструкторы и статические конструкторы должны иметь одинаковые имена с именами непосредственного заключающего класса. Все другие члены должны иметь имена, отличающиеся от имен непосредственного заключающего класса.
* Имена константы, поля, свойства, события или типа должны отличаться от имен всех других членов, объявленных в том же классе.
* Имя метода должно отличаться от всех других объявленных в том же классе имен, не относящихся к методам. Кроме того, подпись (§3.6) метода должна отличаться от подписей всех других методов, объявленных в этом же классе, а два метода, объявленные в одном и том же классе, не могут иметь подписи, отличающиеся только параметрами ref и out.
* Сигнатура конструктора экземпляра должна отличаться от сигнатур всех других конструкторов экземпляров, объявленных в том же классе, а два конструктора экземпляра, объявленных в одном классе, не могут иметь сигнатуры, отличающиеся только словами ref и out.
* Сигнатура индексатора должна отличаться от сигнатур всех других индексаторов, объявленных в этом же классе.
* Сигнатура оператора должна отличаться от сигнатур всех других операторов, объявленных в том же классе.

Унаследованные члены типа класса (§10.3.3) не являются частью области объявления класса. Поэтому производный класс может объявлять член с именем или сигнатурой, совпадающей с именем унаследованного члена (что приводит к скрытию унаследованного члена).

### Тип экземпляра

Каждое объявление класса имеет связанный привязанный тип (§4.4.3), тип экземпляра. Для объявления универсального класса тип экземпляра формируется путем создания сформированного типа (§4.4) на основе объявления типа с каждым предоставленным аргументом типа, являющимся соответствующим параметром типа. Поскольку тип экземпляра использует параметры типа, он может быть использован только в том случае, когда параметры типа находятся в его области, т. е. в рамках объявления класса. Типом экземпляра является this для кода, написанного внутри объявления класса. Для неуниверсальных классов типом экземпляра является просто объявленный класс. Далее представлены объявления класса с их типами экземпляров.

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

### Члены сформированных типов

Не унаследованные члены сформированного типа получаются путем замещения для каждого параметра типа в объявлении члена соответствующего аргумента типа сформированного типа. Процесс замещения основан на семантическом значении объявлений типа и не является простым текстовым замещением.

Например, в объявлении универсального класса

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) {...}  
}

сформированный тип Gen<int[],IComparable<string>> имеет следующие члены

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) {...}

Типом члена a в объявлении универсального класса Gen является «двумерный массив T», поэтому типом члена a в сформированном выше типе является «двумерный массив одномерного массива int», или int[,][].

В рамках членов функции экземпляра типом this является тип экземпляра (§10.3.1) содержащего объявления.

Все члены универсального класса могут использовать параметры типа из любого включающего класса непосредственно или в качестве части сформированного типа. Когда определенный закрытый сформированный тип (§4.4.2) используется во время выполнения, каждое использование параметра типа заменяется фактическим аргументом типа, предоставляемым сформированному типу. Пример:

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  
 }  
}

### Наследование

Класс наследует члены своего прямого типа базового класса. Наследование означает, что класс неявно содержит все члены его прямого типа базового класса, исключая конструкторы экземпляров, деструкторы и статические конструкторы базового класса. Далее перечислены некоторые важные аспекты наследования.

* Наследование транзитивно. Если C наследуется от B, а B наследуется от A, то C унаследует члены, объявленные и в B, и в A.
* Производный класс расширяет свой прямой базовый класс. Производный класс может добавлять новые члены к своим унаследованным членам, но он не может удалить определение унаследованного члена.
* Конструкторы экземпляров, деструкторы и статические конструкторы не наследуются, но все другие члены, независимо от их объявленной доступности, наследуются (§3.5). Но в зависимости от их объявленной доступности, унаследованные члены могут быть недоступны в производном классе.
* Производный класс может скрывать (§3.7.1.2) унаследованные члены путем объявления новых членов с тем же именем или с той же сигнатурой. Обратите внимание, что скрытие унаследованного члена не приводит к удалению данного члена, оно просто приводит к тому, что этот член становится недоступен непосредственно через производный класс.
* Экземпляр класса содержит набор всех полей экземпляра, объявленных в классе и его базовом классе, и существует неявное преобразование (§6.1.6) из типа производного класса в любой из его типов базового класса. Таким образом, ссылка на экземпляр некоторого производного класса может рассматриваться как ссылка на экземпляр какого-либо из его базовых классов.
* Класс может объявить виртуальные методы, свойства и индексаторы, а производный класс может переопределить реализацию данных членов функций. Это позволяет классам реализовывать полиморфное поведение, при котором действия, выполняемые вызовом члена функции, отличаются в зависимости от типа времени выполнения экземпляра, в котором вызывается член функции.

Унаследованным членом типа сформированного класса являются члены непосредственного типа базового класса (§10.1.4.1), который обеспечивается путем замещения аргументов типа сформированного типа для каждого вхождения соответствующих параметров типа в спецификации базового класса. В свою очередь, данные члены преобразуются путем замещения для каждого параметра типа в объявлении члена соответствующего аргумента типа спецификации базового класса.

class B<U>  
{  
 public U F(long index) {...}  
}

class D<T>: B<T[]>  
{  
 public T G(string s) {...}  
}

В предыдущем примере сформированный тип D<int> имеет неунаследованный член public int G(string s), получаемый путем замещения аргумента типа int на параметр типа T.D<int>, который также имеет наследуемый член из объявления класса B. Данный унаследованный член определяется путем определения типа базового класса B<int[]> из D<int> посредством замещения int для T в спецификации базового класса B<T[]>. Затем в качестве аргумента типа B int[] заменяет U в строке public U F(long index), что приводит к унаследованному члену public int[] F(long index).

### Модификатор new

Для объявления члена класса допускается объявление члена с таким же именем или сигнатурой, что и у унаследованного члена. При этом член производного класса скрывает член базового класса. Скрытие унаследованного члена не считается ошибкой, но приводит к отображению компилятором предупреждения. Чтобы эти предупреждения не появлялись, объявление члена производного класса может включать модификатор new для указания того, что производный член предназначен для скрытия базового члена. Эта тема более подробно рассматривается в §3.7.1.2.

Если модификатор new включен в объявление, не скрывающее унаследованный член, создается предупреждение об этом эффекте. Чтобы не выводить такие предупреждения, следует удалить модификатор new.

### Модификаторы доступа

Объявление члена класса может иметь любой из пяти возможных видов объявленной доступности (§3.5.1): public, protected internal, protected, internal или private. Исключая комбинацию protected internal, указание более одного модификатора доступа является ошибкой времени компилирования. Если объявление члена класса не включает какие-либо модификаторы доступа, предполагается использование модификатора private.

### Составные типы

Типы, используемые в объявлении члена, являются составными типами данного члена. Возможные составные члены — это типы константы, поля, свойства, события или индексатора, тип возвращаемого значения метода или оператора и типы параметров метода, индексатора, оператора или конструктора экземпляров. Составные типы члена должны быть не менее доступны, чем сам член (§3.5.4).

### Статические члены и члены экземпляра

Члены класса являются либо статическими членами, либо членами экземпляра. В общем случае считается, что статические члены принадлежат к типам классов, а члены экземпляра принадлежат к объектам (экземплярам типов классов).

Когда объявление поля, метода, свойства, события, оператора или конструктора содержит модификатор static, оно объявляет статический член. Кроме того, объявление константы или типа неявно объявляет статический член. Статические члены имеют следующие особенности.

* Если на статический член M ссылается в доступе члена (§7.6.4) формы E.M, E должен обозначать тип-контейнера M. Обозначение E экземпляром является ошибкой времени компилирования.
* Статическое поле идентифицирует только одно место хранения для всех экземпляров данного закрытого типа класса. Независимо от количества создаваемых экземпляров данного закрытого типа класса всегда существует только одна копия статического поля.
* Статическая функция-член (метод, свойство, событие, оператор или конструктор) не выполняет операций с конкретным экземпляром. При использовании зарезервированного слова this в подобной функции-члене возникает ошибка компиляции.

Если объявление поля, метода, свойства, события, индексатора, конструктора или деструктора не включает модификатор static, оно объявляет член экземпляра. (Член экземпляра иногда называется нестатическим членом). Члены экземпляра имеют следующие особенности.

* Когда на член экземпляра M ссылаются в доступе члена (§7.6.4) формыE.M, E должен обозначать экземпляр тип-контейнера M. Обозначение E типом является ошибкой времени привязки.
* Каждый экземпляр класса содержит отдельный набор всех полей экземпляра класса.
* Член функции экземпляра (метод, свойство, индексатор, конструктор экземпляра или деструктор) работает на заданном экземпляре класса, и данный экземпляр может быть вызван в качестве this (§7.6.7).

В следующем примере представлены правила доступа к статическим членам и членам экземпляров.

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  
 }  
}

Метод F указывает, что в члене функции экземпляра простое имя (§7.6.2) может использоваться для доступа к членам экземпляров и статическим членам. Метод G указывает, что вызов члена экземпляра через простое имя в статическом члене функции является ошибкой времени компилирования. Метод Main указывает, что в доступе члена (§7.6.4) члены экземпляра должны вызываться через экземпляры, а статические члены должны вызываться через типы.

### Вложенные типы

Тип, объявленный в рамках объявления класса или структуры, называется вложенным типом. Тип, объявленный в рамках единицы компиляции или пространства имен, называется невложенным типом.

В этом примере

using System;

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

В этом примере класс B является вложенным типом, так как он объявлен в рамках класса A, а класс A является невложенным типом, так как он объявлен в рамках единицы компиляции.

#### Полные имена

Полным именем (§3.8.1) для вложенного типа является S.N, где S является полным именем типа, в котором объявлен тип N.

#### Объявленная доступность

Невложенные типы могут иметь объявленную доступность public или internal, по умолчанию используется internal. Вложенные типы тоже могут иметь эти виды объявленной доступности, плюс один или более видов объявленной доступности, в зависимости от того, является ли тип-контейнер классом или структурой.

* Вложенный тип, объявленный в классе, может иметь любой из пяти видов объявленной доступности (public, protected internal protected, internal или private), подобно другим членам класса, в нем по умолчанию используется private.
* Вложенный тип, объявленный в структуре, может иметь любой из трех видов объявленной доступности (public, internal или private), подобно другим членам структуры, в нем по умолчанию используется объявленная доступность private.

Пример:

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 {...} }  
}

В этом примере объявляется частный вложенный класс Node.

#### Скрытие

Вложенный тип может скрывать (§3.7.1) базовый член. В объявлениях вложенных типов разрешен модификатор new, так что скрытие можно выразить явно. Пример:

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();  
 }  
}

показан вложенный класс M, скрывающий метод M, определенный в Base.

#### Доступ this

Вложенный тип и его тип-контейнер не имеют особого отношения к доступу this (§7.6.7). В частности, this внутри вложенного типа нельзя использовать для ссылки на члены экземпляра типа-контейнера. В случае, если вложенному типу требуется доступ к членам экземпляра типа-контейнера, доступ может быть выполнен предоставлением this для экземпляра типа-контейнера в качестве аргумента конструктора для вложенного типа. Пример.

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();  
 }  
}

В этом примере показан этот способ. Экземпляр C создает экземпляр Nested и передает свой собственный this конструктору Nested, чтобы предоставить последовательный доступ к членам экземпляра C.

#### Доступ к частным и защищенным членам типа-контейнера

Вложенный тип имеет доступ ко всем членам, к которым есть доступ из типа-контейнера, в том числе к членам типа-контейнера с объявленной доступностью private и protected. Пример:

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();  
 }  
}

показан класс C, содержащий вложенный класс Nested. Внутри Nested метод G вызывает статический метод F, определенный в классе C; метод F имеет объявленную доступность private.

Вложенный тип может также иметь доступ к защищенным членам, определенным в базовом типе его типа-контейнера. В этом примере

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();  
 }  
}

вложенный класс Derived.Nested обращается к методу F с доступностью protected, определенному в Base — базовом классе класса Derived, — путем вызова через экземпляр класса Derived.

#### Вложенные типы в универсальных классах

Объявление универсального типа может содержать объявления вложенных типов. Параметры типа включающего класса можно использовать внутри вложенных типов. Объявление вложенного типа может содержать дополнительные параметры типа, применяемые только ко вложенному типу.

Каждое объявление типа, содержащееся внутри объявления универсального типа, неявно является объявлением универсального типа. При записи обращения к типу, вложенному в универсальный тип, сформированный тип-контейнер, включая его аргументы типа, должен быть именован. Однако изнутри внешнего класса вложенный тип можно использовать без уточнения; тип экземпляра внешнего класса может быть неявно использован при формировании вложенного типа. В следующем примере показаны три разных правильных способа обращения к сформированному типу, созданному из Inner; первые два способа эквивалентны.

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  
 }  
}

Хотя это плохой стиль программирования, параметр типа во вложенном типе может скрывать член или параметр типа, объявленный во внешнем типе:

class Outer<T>  
{  
 class Inner<T> // Valid, hides Outer’s T  
 {  
 public T t; // Refers to Inner’s T  
 }  
}

### Зарезервированные имена членов

Для упрощения реализации среды выполнения базового C# для каждого объявления исходного члена, являющегося свойством, событием или индексатором, в реализации должны быть зарезервированы две сигнатуры методов на основе вида объявления члена, его имени и типа. Объявление в программе члена, сигнатура которого совпадает с одной из этих зарезервированных сигнатур, даже если базовая реализация среды выполнения не использует эти резервирования, является ошибкой времени компиляции.

Зарезервированные имена не вводят объявления, и таким образом не участвуют в поиске членов. Но подписи зарезервированного метода, связанные с объявлением, все же участвуют в наследовании (§10.3.3) и могут быть скрыты с помощью модификатора new (§10.3.4).

Резервирование этих имен служит трем целям:

* разрешить базовой реализации использовать обычный идентификатор в качестве имени метода для получения или установки доступа к средствам языка C#;
* разрешить другим языкам возможность взаимодействия, используя обычный идентификатор в качестве имени метода для получения или установки доступа к средствам языка C#;
* обеспечение того, чтобы исходный код, принятый одним соответствующим компилятором, был принят и другим, сделав особенности зарезервированных имен членов согласованными по всем реализациям C#.

Объявление деструктора (§10.13) также вызывает резервирование подписи (§10.3.9.4).

#### Имена членов, зарезервированные для свойств

Для свойства P (§10.7) типа T зарезервированы следующие подписи:

T get\_P();  
void set\_P(T value);

Зарезервированы обе сигнатуры, даже если свойство доступно только для чтения или только для записи.

В этом примере

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());  
 }  
}

класс A определяет доступное только для чтения свойство P, резервируя тем самым сигнатуры для методов get\_P и set\_P. Класс B является производным от A, и скрывает обе эти зарезервированные сигнатуры. Далее показан вывод для вышеуказанного примера.

123  
123  
456

#### Имена членов, зарезервированные для событий

Для события E (§10.8) типа делегата T зарезервированы следующие подписи:

void add\_E(T handler);  
void remove\_E(T handler);

#### Имена членов, зарезервированные для индексаторов

Для индексатора (§10.9) типа T со списком параметров L зарезервированы следующие подписи:

T get\_Item(L);  
void set\_Item(L, T value);

Зарезервированы обе сигнатуры, даже если индексатор доступен только для чтения или только для записи.

Кроме того, имя члена Item является зарезервированным.

#### Имена членов, зарезервированные для деструкторов

Для класса, содержащего деструктор (§10.13), зарезервирована следующая сигнатура:

void Finalize();

## Константы

Константа – это член класса, представляющий постоянное значение: значение, которое может быть вычислено во время компиляции. Объявление константы вводит одну или более констант указанного типа.

constant-declaration:  
attributesopt constant-modifiersopt 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

Объявление константы может включать набор атрибутов (§17), модификатор new (§10.3.4) и допустимое сочетание из четырех модификаторов доступа (§10.3.5). Атрибуты и модификаторы применяются ко всем членам, объявленным объявлением константы. Хотя константы считаются статическими членами, объявление константы не требует и не допускает модификатор static. Неоднократное появление одного и того же модификатора в объявлении константы является ошибкой.

Тип в объявлении константы указывает тип членов, вводимых объявлением. За типом следует список деклараторов констант, каждый из которых вводит новый член. Декларатор константы состоит из идентификатора, именующего член, за которым следует лексема «=», а затем константное выражение (§7.19), задающее значение члена.

Тип, указанный в объявлении константы, должен быть типом sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, перечисляемым типом или ссылочным типом. Каждое константное выражение должно давать значение конечного типа или типа, который может быть неявно преобразован к конечному типу (§6.1).

Тип константы должен быть, по крайней мере, таким же доступным, как сама константа (§3.5.4).

Значение константы получается в выражении с помощью простого имени (§7.6.2) или доступа к члену (§7.6.4).

Константа сама может участвовать в константном выражении. Таким образом, константу можно использовать в любой конструкции, где требуется константное выражение. Примеры таких конструкций включают метки case, операторы goto case, объявления членов enum, атрибуты и другие объявления констант.

Как описано в §7.19, константное выражение является выражением, которое может быть полностью вычислено во время компиляции. Поскольку единственный способ создать отличное от NULL значение ссылочного типа, отличное от string, – применить оператор new, а оператор new не допускается в константном выражении, единственным возможным значением для констант ссылочных типов, отличных от string, является null.

Если для значения константы требуется символическое имя, а тип этого значения не допускается в объявлении константы, или если значение невозможно вычислить во время компиляции посредством константного выражения, вместо этого можно использовать поле readonly (§10.5.2).

Объявление константы, объявляющее несколько констант, эквивалентно нескольким объявлениям одиночных констант с теми же атрибутами, модификаторами и типом. Пример

class A  
{  
 public const double X = 1.0, Y = 2.0, Z = 3.0;  
}

равнозначно

class A  
{  
 public const double X = 1.0;  
 public const double Y = 2.0;  
 public const double Z = 3.0;  
}

Константы могут зависеть от других констант внутри одной программы, если только зависимости не являются циклическими. Компилятор автоматически располагает вычисления объявлений констант в соответствующем порядке. В этом примере

class A  
{  
 public const int X = B.Z + 1;  
 public const int Y = 10;  
}

class B  
{  
 public const int Z = A.Y + 1;  
}

компилятор сначала вычисляет A.Y, затем B.Z и наконец A.X, создавая значения 10, 11 и 12. Объявления констант могут зависеть от констант из других программ, но такие зависимости возможны только в одном направлении. В вышеприведенном примере: если бы A и B были объявлены в отдельных программах, для A.X было бы возможно зависеть от B.Z, но тогда B.Z не мог бы одновременно зависеть отA.Y.

## Поля

Поле – это член, представляющий переменную, связанную с объектом или классом. Объявление поля вводит одно или более полей заданного типа.

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

Объявление поля может включать набор атрибутов (§17), модификатор new (§10.3.4), допустимое сочетание из четырех модификаторов доступа (§10.3.5) и модификатор static (§10.5.1). Кроме того, объявление поля может включать модификатор readonly (§10.5.2) или volatile (§10.5.3), но не оба. Атрибуты и модификаторы применяются ко всем членам, объявленным объявлением поля. Неоднократное появление одного и того же модификатора в объявлении поля является ошибкой.

Тип в объявлении поля указывает тип членов, вводимых объявлением. За типом следует список деклараторов переменных, каждый из которых вводит новый член. Декларатор переменной состоит из идентификатора, именующего член, за которым необязательно следуют лексема «=» и инициализатор переменной (§10.5.5), задающий начальное значение члена.

Тип поля должен быть, по крайней мере, таким же доступным, как само поле (§3.5.4).

Значение поля получается в выражении с помощью простого имени (§7.6.2) или доступа к члену (§7.6.4). Значение поля с доступом не только для чтения изменяется с помощью присваивания (§7.17). Значение поля с доступом не только для чтения можно и получать, и изменять с помощью операторов постфиксного увеличения и уменьшения (§7.6.9) и операторов префиксного увеличения и уменьшения (§7.7.5).

Объявление поля, объявляющее несколько полей, эквивалентно нескольким объявлениям одиночных полей с теми же атрибутами, модификаторами и типом. Пример

class A  
{  
 public static int X = 1, Y, Z = 100;  
}

равнозначно

class A  
{  
 public static int X = 1;  
 public static int Y;  
 public static int Z = 100;  
}

### Статические поля и поля экземпляров

Если объявление поля включает модификатор static, поля, введенные этим объявлением, являются статическими полями. Если нет модификатора static, введенные объявлением поля являются полями экземпляров. Статические поля и поля экземпляров – это два из нескольких видов переменных (§5), поддерживаемых языком C#, и иногда их называют статическими переменными и переменными экземпляров соответственно.

Статическое поле не является частью конкретного экземпляра, а используется совместно всеми экземплярами закрытого типа (§4.4.2). Независимо от того, сколько создано экземпляров закрытого типа класса, всегда существует только одна копия статического поля для соответствующей области приложения.

Пример:

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  
 }  
}

Поле экземпляра принадлежит экземпляру. В частности, каждый экземпляр класса содержит отдельный набор всех полей экземпляров этого класса.

Если к полю производится обращение через доступ к члену (§7.6.4) вида E.M, если M является статическим полем, E должно означать тип, содержащий M, а если M является полем экземпляра, E должно означать экземпляр типа, содержащего M.

Различия между статическими членами и членами экземпляров рассматриваются в разделе §10.3.7.

### Поля только для чтения

Если объявление поля включает модификатор readonly, поля, введенные этим объявлением, являются полями только для чтения. Прямые присваивания полям только для чтения могут быть только как часть этого объявления или в конструкторе экземпляров, либо в статическом конструкторе в этом же классе. (Присваивание полям только для чтения в этих контекстах можно делать многократно). А именно, прямые присваивания полю readonly разрешены только в следующих контекстах:

* в деклараторе переменной, который вводит это поле (включением в объявление инициализатора переменной);
* для поля экземпляра в конструкторах экземпляров того класса, который содержит объявление поля; для статического поля в статическом конструкторе того класса, который содержит объявление поля. Это также единственные контексты, в которых допускается передача поля readonly в качестве параметра out или ref.

Попытка присваивания полю readonly или передачи его в качестве параметра out или ref в любом другом контексте является ошибкой времени компиляции.

#### Использование статических полей только для чтения вместо констант

Поле static readonly полезно, если требуется символическое имя для значения константы, но тип значения недопустим в объявлении const, или если значение не может быть вычислено во время компиляции. В этом примере

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;  
 }  
}

члены Black, White, Red, Green и Blue нельзя объявить как const, так как их значения нельзя вычислить во время компиляции. Но объявление их как static readonly приводит почти к такому же результату.

#### Отслеживание версий констант и статических полей только для чтения

Константы и поля только для чтения имеют разную бинарную семантику отслеживания версий. Если выражение ссылается на константу, значение этой константы получается во время компиляции, но если выражение ссылается на поле только для чтения, значение этого поля известно только во время выполнения. Рассмотрим приложение, состоящее из двух отдельных программ.

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);  
 }  
 }  
}

Пространства имен Program1 и Program2 обозначают две программы, компилируемые отдельно. Поскольку Program1.Utils.X объявлено как статическое поле только для чтения, выходное значение оператора Console.WriteLine не известно во время компиляции, но получается во время выполнения. Таким образом, если значение X изменяется и Program1 компилируется повторно, оператор Console.WriteLine выведет новое значение, даже если Program2 не компилируется повторно. Но если бы X был константой, значение X было бы получено при компиляции Program2 и осталось бы независимым от изменений в Program1 до перекомпиляции Program2.

### Поля с модификатором volatile

Если объявление поля включает модификатор volatile, поля, введенные этим объявлением, являются полями с модификатором volatile.

Способы оптимизации, переупорядочивающие инструкции, для полей не-volatile могут привести к непредвиденным и непредсказуемым результатам в многопоточных программах, которые обращаются к полям без синхронизации, такой как предоставляемая оператором блокировки (§8.12). Эти оптимизации могут выполняться компилятором, системой поддержки выполнения или оборудованием. Для полей volatile такие переупорядочивающие оптимизации ограничены.

* чтение поля volatile называется чтением volatile. У чтения volatile имеется «семантика захвата», то есть, оно гарантированно выполняется прежде любых обращений к памяти, расположенных после него в последовательности инструкций;
* запись поля volatile называется записью volatile. У записи volatile имеется «семантика освобождения», то есть, оно гарантированно выполняется после всех обращений к памяти, расположенных до инструкции записи в последовательности инструкций.

Эти ограничения приводят к тому, что все потоки будут видеть записи volatile, выполняемые любым другим потоком, в том порядке, в каком они выполнялись. Не требуется соответствующая реализация для обеспечения единого общего упорядочения записей volatile, представляемого для всех потоков выполнения. Поле volatile должно имеет один из следующих типов:

* ссылочный тип;
* тип byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr или System.UIntPtr;
* перечисляемый тип, имеющий базовый перечисляемый тип byte, sbyte, short, ushort, int или uint.

Пример:

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;  
 }  
 }  
 }  
}

дает на выходе:

result = 143

В этом примере метод Main запускает новый поток, выполняющий метод Thread2. Этот метод сохраняет значение в поле не-volatile с именем result, затем сохраняет true в поле volatile с именем finished. Главный поток ожидает, пока в поле finished не будет установлено значение true, затем считывает поле result. Так как поле finished объявлено как volatile, главный поток должен прочитать значение 143 из поля result. Если бы поле finished не было объявлено как volatile, то сохранение в result могло быть видимым в главном потоке после сохранения в finished, и главный поток мог прочитать значение 0 из поля result. Объявление поля finished как volatile устраняет такую несогласованность.

### Инициализация поля

Начальным значением поля, как статического, так и поля экземпляра, является значение по умолчанию (§5.2) типа поля. Невозможно видеть значение поля до выполнения этой инициализации по умолчанию, и поэтому поле никогда не бывает «неинициализированным». Пример:

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);  
 }  
}

производятся следующие выходные данные

b = False, i = 0

так как и b, и i автоматически инициализированы значениями по умолчанию.

### Инициализаторы переменных

Объявления полей могут включать инициализаторы переменных. Для статических полей инициализаторы переменных соответствуют операторам присвоения, выполняемым во время инициализации класса. Для полей экземпляров инициализаторы переменных соответствуют операторам присваивания, выполняемым при создании экземпляра класса.

Пример:

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);  
 }  
}

производятся следующие выходные данные

x = 1.4142135623731, i = 100, s = Hello

так как присваивание для x происходит при выполнении инициализаторов статического поля, а присваивания для i и s происходят при выполнении инициализаторов поля экземпляра.

Инициализация значения по умолчанию, описанная в §10.5.4, происходит для всех полей, включая поля, имеющие инициализаторы переменных. Таким образом, когда класс инициализируется, все статические поля в этом классе сначала инициализируются своими значениями по умолчанию, а затем выполняются инициализаторы статических полей в текстовом порядке. Аналогично, когда создается экземпляр класса, все поля в этом экземпляре сначала инициализируются своими значениями по умолчанию, а затем выполняются инициализаторы полей экземпляров в текстовом порядке.

Статические поля с инициализаторами переменных можно наблюдать в их состоянии со значениями по умолчанию. Однако такой стиль настоятельно не рекомендуется. Пример:

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);  
 }  
}

В этом примере показывается такое поведение. Несмотря на циклические определения a и b, программа допустима. Она дает на выходе

a = 1, b = 2

так как статические поля a и b инициализированы значением 0 (значение по умолчанию для int) до выполнения их инициализаторов. Когда выполняется инициализатор для a, значение b равно нулю, так что a инициализируется значением 1. Когда выполняется инициализатор для b, значение a уже равно 1, так что b инициализируется значением 2.

#### Инициализация статического поля

Инициализаторы переменных статических полей класса соответствуют последовательности присваиваний, выполняемых в текстовом порядке, в котором они появляются в объявлении класса. Если в классе существует статический конструктор (§10.12), выполнение инициализаторов статического поля происходит непосредственно перед выполнением этого статического конструктора. Иначе инициализаторы статических полей выполняются в зависящий от реализации момент перед первым использованием статического поля этого класса. Пример:

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");  
}

В этом примере на выходе может быть

Init A  
Init B  
1 1

или

Init B  
Init A  
1 1

так как выполнение инициализатора X и инициализатора Y могло произойти в том или другом порядке; они только должны быть выполнены до обращений к этим полям. Но в примере

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");  
}

результатом должно быть:

Init B  
Init A  
1 1

так как правила выполнения статических конструкторов (как определено в §10.12) обеспечивают, что статический конструктор для B (и следовательно инициализаторы статического поля B) должен выполняться до статического конструктора и инициализаторов поля A.

#### Инициализация поля экземпляра

Инициализаторы переменных полей экземпляров класса соответствуют последовательности присваиваний, выполняемых непосредственно при входе в любой из конструкторов экземпляров (§10.11.1) этого класса. Инициализаторы переменных выполняются в том текстовом порядке, в каком они представлены в объявлении класса. Создание экземпляра класса и процесс инициализации описаны далее в разделе §10.11.

Инициализатор переменной для поля экземпляра не может обращаться к создаваемому экземпляру. Таким образом, обращение к this в инициализаторе переменной, как и обращение инициализатора переменной к любому члену экземпляра по простому имени, является ошибкой времени компиляции. В этом примере

class A  
{  
 int x = 1;  
 int y = x + 1; // Error, reference to instance member of this  
}

инициализатор переменной для y приводит к ошибке времени компиляции, так как ссылается на член создаваемого экземпляра.

## Методы

Метод — это член, реализующий вычисление или действие, которое может быть выполнено объектом или классом. Методы объявляются с помощью объявлений методов:

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  
;

Объявление метода может включать набор атрибутов (§17) допустимое сочетание любых из четырех модификаторов доступа (§10.3.5), а также модификаторы new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) и extern (§10.6.7).

Объявление имеет допустимое сочетание модификаторов, если верно все следующее:

* объявление включает допустимое сочетание модификаторов доступа (§10.3.5);
* объявление не включает один и тот же модификатор несколько раз;
* объявление включает не более одного из следующих модификаторов: static, virtual и override;
* объявление включает не более одного из следующих модификаторов: new и override;
* если объявление включает модификатор abstract, оно не включает ни один из следующих модификаторов: static, virtual, sealed и extern;
* если объявление включает модификатор private, оно не включает ни один из следующих модификаторов: virtual, override и abstract;
* если объявление включает модификатор sealed, оно также включает модификатор override;
* если объявление включает модификатор partial, оно не включает ни один из следующих модификаторов: new, public, protected, internal, private, virtual, sealed, override, abstract и extern.

Метод с модификатором async является асинхронной функцией, которая следует правилам, описанным в разделе §10.14.

Тип возвращаемого значения в объявлении метода указывает тип значения, вычисляемого и возвращаемого методом. Если метод не возвращает значение, типом возвращаемого значения является void. Если объявление включает модификатор partial, типом возвращаемого значения должен быть void.

Имя члена указывает имя метода. Если только метод не является членом явной реализации интерфейса (§13.4.1), имя члена – это просто идентификатор. Для члена явной реализации интерфейса имя члена состоит из типа интерфейса, за которым следует «.» и идентификатор.

Необязательный список параметров типа указывает параметры типа метода (§10.1.3). Если список параметров типа указан, метод является универсальным методом. Если у метода есть модификатор extern, список параметров типа указывать нельзя.

Необязательный список формальных параметров указывает параметры метода (§10.6.1).

Необязательные предложения ограничений параметров типов указывают ограничения по отдельным параметрам типов (§10.1.5) и могут быть указаны, только если предоставлен также список параметров типов, а у метода нет модификатора override.

Тип возвращаемого значения и каждый из типов в списке формальных параметров метода должен быть, по крайней мере, так же доступным, как сам метод (см. §3.5.4).

Для методов abstract и extern тело метода состоит просто из точки с запятой. Для методов partial тело метода может состоять либо из точки с запятой, либо из блока. Для всех других методов тело метода состоит из блока, в котором указаны операторы для исполнения при вызове метода.

Если тело метода состоит из точки с запятой, объявление не может включать модификатор async.

Имя, список параметров типа и список формальных параметров метода определяют сигнатуру (§3.6) метода. А именно, сигнатура метода состоит из его имени, числа параметров типов и числа, модификаторов и типов его формальных параметров. Для этих целей каждый параметр типа метода, встречающийся в типе формального параметра, идентифицируется не по своему имени, а по своему порядковому номеру в списке аргументов типа метода. Тип возвращаемого значения не является частью сигнатуры метода, как и имена параметров типов и формальных параметров.

Имя метода должно отличаться от всех других объявленных в том же классе имен, не относящихся к методам. Кроме того, подпись метода должна отличаться от подписей всех других методов, объявленных в этом же классе, а два метода, объявленные в одном и том же классе, не могут иметь подписи, отличающиеся только параметрами ref и out.

Параметры типа метода находятся в области всего объявления метода, и могут использоваться для формирования типов в этой области в типе возвращаемого значения, теле метода и в предложениях ограничений типов параметров, но не в атрибутах.

Все формальные параметры и параметры типов должны иметь разные имена.

### Параметры метода

Параметры метода, если они имеются, объявляются списком формальных параметров метода.

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

Список формальных параметров состоит из одного или более параметров, разделенных запятыми, из которых только последний может быть массивом параметров.

Фиксированный параметр состоит из необязательного набора атрибутов (§17), необязательного модификатора ref, out или this, типа, идентификатора и необязательного аргумента по умолчанию. Каждый фиксированный параметр объявляет параметр данного типа с данным именем. Модификатор this определяет метод как метод расширения и допускается только в первом параметре статического метода. Методы расширения описаны далее в §10.6.9.

Фиксированный параметр с аргументом по умолчанию называется необязательным параметром, в то время как фиксированный параметр без аргумента по умолчанию является обязательным параметром. Обязательный параметр не может стоять после необязательного параметра в списке формальных параметров.

Параметры ref и out не могут иметь аргумент по умолчанию. Выражением в аргументе по умолчанию должно быть одно из следующих:

* константное выражение;
* выражение вида new S(), где S является типом значения;
* выражение вида default(S), где S является типом значения;

Выражение должно неявно преобразовываться идентификацией преобразования, которое может иметь значение NULL, в тип параметра.

Если необязательные параметры встречаются в определяющем объявлении разделяемого метода (§10.2.7), в явной реализации интерфейса члена интерфейса (§13.4.1) или в объявлении индексатора единственного параметра (§10.9), компилятор должен создать предупреждение, поскольку эти члены никогда не могут вызываться способом, в котором аргументы могут отсутствовать.

Массив параметров состоит из необязательного набора атрибутов (§17), модификатора params, типа массива и идентификатора. Массив параметров объявляет единственный параметр заданного типа массива с данным именем. Тип массива массива параметров должен быть типом одномерного массива (§12.1). При вызове метода массив параметров допускает указание либо одного аргумента с данным типом массива, либо ноль или более аргументов типа элемента массива. Параметры массивов описаны далее в §10.6.1.4.

Массив параметров может присутствовать после необязательного параметра, но не может иметь значение по умолчанию – отсутствие аргументов для массива параметров приведет вместо этого к созданию пустого массива.

В двух следующих примерах демонстрируются разные виды параметров.

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  
) { }

В списке формальных параметров для M i является обязательным параметром ref, d является обязательным параметром по значению, b, s, o и t являются необязательными параметрами по значению, и a является массивом параметров.

Объявление метода создает отдельную область объявлений для параметров, параметров типов и локальных переменных. Имена вводятся в эту область объявлений списком параметров типов и списком формальных параметров метода, и объявлениями локальных переменных в блоке метода. Одно и то же имя для двух членов области объявлений метода является ошибкой. Содержание элементов с одним и тем же именем в области объявлений метода и в области объявлений локальных переменных вложенной области объявлений является ошибкой.

Вызов метода (§7.6.5.1) создает определенную для этого вызова копию формальных параметров и локальных переменных метода, а список аргументов вызова присваивает значения или ссылки на переменные вновь созданным формальным параметрам. Внутри блока метода к формальным параметрам можно обращаться по их идентификаторам в выражениях простого имени (§7.6.2).

Есть четыре вида формальных параметров:

* параметры по значению, объявляемые без модификаторов;
* параметры по ссылке, объявляемые с модификатором ref;
* выходные параметры, объявляемые с модификатором out;
* массивы параметров, объявляемые с модификатором params.

Как описано в §3.6, модификаторы ref и out являются частью подписи метода, а модификатор params – нет.

#### Параметры по значению

Параметр, объявленный без модификаторов, является параметром по значению. Параметр по значению соответствует локальной переменной, получающей свое начальное значение из соответствующего аргумента, предоставленного во время вызова метода.

Если формальный параметр является параметром по значению, соответствующий аргумент в вызове метода должен быть выражением, неявно преобразуемым (§6.1) в тип формального параметра.

Методу разрешено присваивать новые значения параметру по значению. Такие присваивания влияют только на локальное место хранения, представленное параметром по значению – они не влияют на фактический аргумент, заданный при вызове метода.

#### Параметры ссылок

Параметр, объявленный с модификатором ref, является параметром по ссылке. В отличие от параметра по значению, параметр по ссылке не создает новое место хранения. Вместо этого параметр по ссылке представляет то же самое место хранения, что и переменная, заданная в качестве аргумента при вызове метода.

Если формальный параметр является параметром по ссылке, соответствующий аргумент в вызове метода должен состоять из зарезервированного слова ref, за которым следует ссылка на переменную (§5.3.3) того же типа, что и формальный параметр. Переменная должна быть определенно назначенной до того, как ее можно будет передать в качестве параметра по ссылке.

Внутри метода параметр по ссылке всегда считается определенно назначенным.

У метода, объявленного как итератор (§10.14), не может быть параметров по ссылке.

Пример:

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);  
 }  
}

производятся следующие выходные данные

i = 2, j = 1

Для вызова метода Swap в классе Main, x представляет i, а y представляет j. Таким образом, результатом вызова является обмен значений i и j.

В методе, принимающем параметры по ссылке, несколько имен могут представлять одно и то же место хранения. В этом примере

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);  
 }  
}

вызов F из G передает ссылку на s одновременно для a и b. Таким образом, при этом вызове все имена s, a и b ссылаются на одно и то же место хранения, а все три присваивания изменяют поле экземпляра s.

#### Выходные параметры

Параметр, объявленный с модификатором out, является выходным параметром. Подобно параметру по ссылке, выходной параметр не создает новое место хранения. Вместо этого выходной параметр представляет то же самое место хранения, что и переменная, заданная в качестве аргумента при вызове метода.

Если формальный параметр является выходным параметром, соответствующий аргумент в вызове метода должен состоять из зарезервированного слова out, за которым следует ссылка на переменную (§5.3.3) того же типа, что и формальный параметр. Переменной не требуется быть определенно назначенной, прежде чем ее можно будет передать в качестве выходного параметра, но вслед за вызовом, в котором переменная была передана в качестве выходного параметра, эта переменная считается определенно назначенной.

Внутри метода, точно так же, как и локальная переменная, выходной параметр вначале считается неназначенным, и должен быть определенно назначен до использования его значения.

Каждый выходной параметр метода должен быть определенно назначен до возврата из метода.

Метод, объявленный как частичный (§10.2.7) или как итератор (§10.14), не может иметь выходные параметры.

Выходные параметры обычно используются в методах, создающих несколько возвращаемых значений. Пример:

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);  
 }  
}

Далее показан вывод для вышеуказанного примера.

c:\Windows\System\  
hello.txt

Обратите внимание, что переменным dir и name может быть не сделано присваивание до их передачи в SplitPath, но они считаются определенно назначенными вслед за вызовом.

#### Массивыпараметров

Параметр, объявленный с модификатором params, является массивом параметров. Если список формальных параметров включает массив параметров, тот должен быть последним параметром в списке и должен иметь тип одномерного массива. Например, типы string[] и string[][] могут использоваться в качестве типа массива параметров, а тип string[,] — не может. Невозможно объединить модификатор params с модификаторами ref и out.

Массив параметров позволяет указывать аргументы одним из двух способов при вызове метода:

* Аргумент, заданный для массива параметров, может быть единственным выражением, неявно преобразуемым (§6.1) в тип массива параметров. В этом случае использование массива параметров не отличается от использования параметра значения.
* Либо в вызове могут быть заданы нуль или несколько аргументов для массива параметров, где каждый аргумент является выражением, неявно преобразуемым (§6.1) в тип элемента массива параметров. В этом случае при вызове создается экземпляр типа массива параметров с длиной, соответствующей числу аргументов, происходит инициализация элементов экземпляра массива с помощью указанных значений аргументов и в качестве фактического аргумента используется этот только что созданный экземпляр массива.

Кроме разрешения переменного числа аргументов при вызове, массив параметров совершенно эквивалентен параметру по значению (§10.6.1.1) того же типа.

Пример:

using System;

class Test  
{  
 static void F(params int[] args) {  
 Console.Write("Array contains {0} elements:", args.Length);  
 foreach (int i in args)   
 Console.Write(" {0}", i);  
 Console.WriteLine();  
 }

static void Main() {  
 int[] arr = {1, 2, 3};  
 F(arr);  
 F(10, 20, 30, 40);  
 F();  
 }  
}

производятся следующие выходные данные

Array contains 3 elements: 1 2 3  
Array contains 4 elements: 10 20 30 40  
Array contains 0 elements:

При первом вызове F просто передает массив a в качестве параметра по значению. При втором вызове F автоматически создает четырехэлементный массив int[] с заданными значениями элементов и передает этот массив в качестве параметра по значению. Аналогично, третий вызов F создает массив из нуля элементов int[] и передает этот экземпляр в качестве параметра по значению. Второй и третий вызовы полностью эквивалентны следующему коду:

F(new int[] {10, 20, 30, 40});  
F(new int[] {});

При выполнении разрешения перегрузки метод с массивом параметров может применяться либо в его обычном виде, либо в расширенном виде (§7.5.3.1). Расширенный вид метода доступен, если только неприменим обычный вид, и если только в том же типе уже не объявлен применимый метод с такой же сигнатурой, как у расширенного вида.

Пример:

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);  
 }  
}

производятся следующие выходные данные

F();  
F(object[]);  
F(object,object);  
F(object[]);  
F(object[]);

В этом примере два из возможных расширенных видов метода с массивом параметров уже включены в класс как обычные методы. Поэтому эти расширенные виды не рассматриваются при выполнении разрешения перегрузки, а первый и третий вызовы метода таким образом выбирают обычные методы. Если в классе объявляют метод с массивом параметров, нередко включают и некоторые из расширенных видов в качестве обычных методов. При этом становится возможным избежать выделения экземпляра массива, которое происходит при вызове расширенного вида метода с массивом параметров.

Если массив параметров имеет тип object[], возникает потенциальная неоднозначность между нормальной и расширенной формами метода для одного параметра типа object. Причиной неоднозначности является то, что object[] сам может неявно преобразовываться в тип object. Неоднозначность не ведет к возникновению проблем и при необходимости может быть разрешена с помощью приведения.

Пример:

using System;

class Test  
{  
 static void F(params object[] args) {  
 foreach (object o in args) {  
 Console.Write(o.GetType().FullName);  
 Console.Write(" ");  
 }  
 Console.WriteLine();  
 }

static void Main() {  
 object[] a = {1, "Hello", 123.456};  
 object o = a;  
 F(a);  
 F((object)a);  
 F(o);  
 F((object[])o);  
 }  
}

производятся следующие выходные данные

System.Int32 System.String System.Double  
System.Object[]  
System.Object[]  
System.Int32 System.String System.Double

При первом и последнем вызове функции F применима нормальная форма F, поскольку существует неявное преобразование из типа аргумента к типу параметра (оба имеют тип object[]). Таким образом, в результате разрешения перегрузки выбирается нормальная форма F, а аргумент передается как регулярный параметр значения. Во втором и в третье вызовах обычная форма F неприменима, поскольку не существует неявного преобразования из типа аргумента в тип параметра (тип object не может быть неявно преобразован в тип object[]). Однако в этом случае применима расширенная форма F, которая выбирается в результате разрешения перегрузки. В результате вызовом создается одноэлементный массив object[], и единственный элемент массива инициализируется данным значением аргумента (который сам является ссылкой на object[]).

### Статические методы и методы экземпляра

Статический метод объявляется с помощью модификатора static. Если модификатор static отсутствует, метод называется методом экземпляра.

Статический метод не выполняет операций с конкретным экземпляром. При использовании зарезервированного слова this в статическом методе возникает ошибка компиляции.

Метод экземпляра выполняет операции с конкретным экземпляром класса, обращение к которому осуществляется с помощью зарезервированного слова this (§7.6.7).

При доступе к члену (§7.6.4) с использованием ссылки на метод в форме E.M для статического метода M параметр E должен обозначать тип, содержащий M, а для метода экземпляра M E должен обозначать экземпляр типа, содержащего M.

Различия между статическими членами и членами экземпляров рассматриваются в разделе §10.3.7.

### Виртуальные методы

Если объявление метода экземпляра содержит модификатор virtual, метод является виртуальным методом. Если модификатор virtual отсутствует, метод называется невиртуальным методом.

Реализация невиртуального метода инвариантна и одинакова вне зависимости от того, вызывается ли метод для экземпляра класса, в котором он объявлен, или для экземпляра производного класса. В отличие от этого, реализация виртуального метода может быть заменена производными классами. Процесс замены реализации унаследованного виртуального метода называется переопределением этого метода (§10.6.4).

При вызове виртуального метода тип времени выполнения экземпляра, для которого осуществляется вызов, определяет фактическую реализацию вызываемого метода. При вызове невиртуального метода определяющим фактором является тип времени компиляции экземпляра. Говоря точнее, если метод N вызывается со списком аргументов A в экземпляре с типом времени компиляции C и с типом времени выполнения R (где R может быть C или экземпляром класса, производного от C), вызов обрабатывается следующим образом.

* Сначала к C, N и A применяется разрешение перегрузки для выбора конкретного метода M из набора методов, объявленных и унаследованных классом C. Это описано в §7.6.5.1.
* Затем, если M является невиртуальным методом, вызывается метод M.
* В противном случае M является виртуальным методом, и вызывается старшая производная реализация метода M по отношению к R.

Для каждого виртуального метода, объявленного в классе или унаследованного им, существует старшая производная реализация метода по отношению к этому классу. Старшая производная реализация виртуального метода M по отношению к классу R определяется следующим образом.

* Если R содержит представляющее объявление virtual метода M, это является старшей производной реализацией M.
* В противном случае, если R содержит объявление override метода M, это является старшей производной реализацией M.
* В противном случае старшей производной реализацией метода M по отношению к R является старшая производная реализация M по отношению к прямому базовому классу R.

В следующем примере показаны различия между виртуальными и невиртуальными методами.

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();  
 }  
}

В этом примере класс A представляет невиртуальный метод F и виртуальный метод G. Класс B представляет новый невиртуальный метод F, скрывая тем самым унаследованную функцию F, а также переопределяет унаследованный метод G. Далее показан вывод для вышеуказанного примера.

A.F  
B.F  
B.G  
B.G

Обратите внимание, что оператор a.G() вызывает метод B.G, а не A.G. Это связано с тем, что фактически вызываемая реализация метода определяется типом времени выполнения (B), а не типом времени компиляции экземпляра (A).

Поскольку методам разрешено скрывать унаследованные методы, класс может содержать несколько виртуальных методов с одной и той же сигнатурой. При этом не возникает проблема неоднозначности, поскольку видимым является только старший производный метод. В этом примере

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();  
 }  
}

классы C и D содержат два виртуальных метода с одинаковыми подписями, один из которых представлен классом A а второй — классом C. Метод, представляемый классом C, скрывает метод, наследуемый от A. Таким образом, объявление переопределения в классе D переопределяет метод, представленный классом C. При этом в классе D невозможно переопределить метод, представленный классом A. Далее показан вывод для вышеуказанного примера.

B.F  
B.F  
D.F  
D.F

Обратите внимание, что можно вызвать скрытый виртуальный метод посредством обращения к экземпляру класса D с помощью младшего производного типа, в котором этот метод не скрыт.

### Переопределяющие методы

Если объявление метода экземпляра содержит модификатор override, метод является переопределяющим методом. Такой метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Объявление виртуального метода представляет новый метод. Объявление переопределяющего метода уточняет существующий виртуальный метод, предоставляя его новую реализацию.

Метод, переопределяемый с помощью объявления override, называется переопределенным базовым методом. Для переопределяющего метода M, объявленного в классе C, переопределенный базовый метод определяется посредством проверки всех базовых для C типов класса. Проверка начинается с прямого базового типа класса C и продолжается для каждого последующего прямого базового типа класса. Проверка продолжается до тех пор, пока не будет найден базовый тип класса, содержащий хотя бы один доступный метод M, подпись которого после замены аргументов типа соответствует заданной. Переопределенный базовый метод считается доступным, если он объявлен как public, protected, protected internal или объявлен как internal в той же программе, что и C.

Если для объявления переопределения не выполняются все следующие условия, возникает ошибка времени компиляции.

* Может быть найден переопределенный базовый метод (см. выше).
* Существует только один переопределенный базовый метод. Это ограничение влияет только в том случае, если тип базового класса представляет собой сформированный тип, в котором сигнатуры двух методов после замены аргументов типа совпадают.
* Переопределенный базовый является виртуальным, абстрактным или переопределяющим методом. Другими словами, переопределенный базовый метод не может быть статическим или невиртуальным.
* Переопределенный базовый метод не является запечатанным методом.
* Переопределяющий метод и переопределенный базовый метод имеют одинаковые типы возвращаемых данных.
* Для объявления переопределения и переопределенного базового метода объявлен одинаковый уровень доступа. Другими словами, объявление переопределения не может изменять доступность виртуального метода. Однако если переопределенный базовый метод объявлен как protected internal в сборке, отличной от той, в которой содержится переопределяющий метод, для последнего следует объявить уровень доступа protected.
* Объявление переопределения не содержит предложений\_ограничений\_параметров\_типа. Все ограничения наследуются из переопределенного базового метода. Обратите внимание, что ограничения, являющиеся параметрами типа в переопределенном методе могут быть заменены аргументами типа в унаследованном ограничении. Это может привести к ограничениям, не являющимся законными при явном задании, таким как типы значений или запечатанные типы.

В следующем примере показано применение правил переопределения для универсальных классов.

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>  
}

Объявление переопределения может иметь доступ к переопределенному базовому методу с помощью доступа base (§7.6.8). В этом примере

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);  
 }  
}

с помощью метода base.PrintFields() класса B вызывается метод PrintFields, объявленный в классе A. Доступ base отключает механизм виртуального вызова и рассматривает базовый метод просто как невиртуальный метод. Если записать вызов в классе B как ((A)this).PrintFields(), это приведет к рекурсивному вызову метода PrintFields, объявленного в классе B, а не в классе A, поскольку метод PrintFields является виртуальным, а типом времени выполнения для ((A)this) является B.

Переопределение другого метода возможно только посредством включения модификатора override. Во всех других случаях метод с сигнатурой, совпадающей с сигнатурой унаследованного метода, просто скрывает унаследованный метод. В этом примере

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 public virtual void F() {} // Warning, hiding inherited F()  
}

метод F класса B не содержит модификатор override и, следовательно, не переопределяет метод F класса A. В этом случае метод F класса B скрывает метод класса A, и отображается предупреждение, поскольку объявление не содержит модификатор new.

В этом примере

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  
}

метод F класса B скрывает виртуальный метод F, унаследованный из класса A. Поскольку для нового метода F класса B объявлен уровень доступа private, область его действия распространяется только на тело класса B и не включает класс C. Таким образом, объявление метода F в классе C может переопределять метод F унаследованный из класса A.

### Запечатанные методы

Если объявление метода экземпляра содержит модификатор sealed, метод является запечатанным методом. Если объявление метода экземпляра включает модификатор sealed, оно должно также включать модификатор override. Использование модификатора sealed предотвращает последующее переопределение метода в производном классе.

Пример:

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");  
 }   
}

класс B предоставляет два переопределяющих метода: метод F с модификатором sealed и метод G без него. Использование модификатораmodifier sealed в классе B препятствует дальнейшему переопределению метода F в классе C.

### Абстрактные методы

Если объявление метода экземпляра содержит модификатор abstract, метод является абстрактным методом. Хотя абстрактный метод неявно является также виртуальным методом, он не может иметь модификатора virtual.

Объявление абстрактного метода представляет новый виртуальный метод, но не предоставляет его реализацию. Вместо этого необходимо предоставить неабстрактные производные классы, предоставляющие собственные реализации этого метода посредством его переопределения. Поскольку абстрактный метод не предоставляет фактическую реализацию, тело абстрактного метода состоит только из точки с запятой.

Объявления абстрактных методов допускаются только в абстрактных классах (§10.1.1.1).

В этом примере

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);  
 }  
}

класс Shape определяет абстрактное представление геометрической фигуры самоокрашивающегося объекта. Метод Paint является абстрактным, поскольку отсутствует значащая реализация по умолчанию. Классы Ellipse и Box представляют собой конкретные реализации класса Shape. Поскольку эти классы являются неабстрактными, требуется, чтобы они переопределяли метод Paint и предоставляли фактическую реализацию.

Если доступ base (§7.6.8) ссылается на абстрактный метод, возникает ошибка времени компиляции. В этом примере

abstract class A  
{  
 public abstract void F();  
}

class B: A  
{  
 public override void F() {  
 base.F(); // Error, base.F is abstract  
 }  
}

при вызове метода base.F() выполняется ссылка на абстрактный метод, что порождает ошибку времени компиляции.

В объявлении абстрактного метода допускается переопределение виртуального метода. Это позволяет абстрактному классу принудительно выполнить повторную реализацию метода в производных классах. При этом исходная реализация метода становится недоступна. В этом примере

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");  
 }  
}

в классе A объявляется виртуальный метод, который переопределяется в классе B абстрактным методом. Последний переопределяется в классе C с использованием собственной реализации.

### Внешние методы

Если объявление метода содержит модификатор extern, метод является внешним методом. Внешние методы обычно реализуются внешне с помощью языков, отличных от C#. Вследствие того, что объявление внешнего метода предоставляет фактическую реализацию, тело метода внешнего метода состоит просто из точки с запятой. Внешний метод не может быть универсальным.

Модификатор extern обычно используется совместно с атрибутом DllImport (§17.5.1), что позволяет реализовывать внешние методы с помощью библиотек динамической компоновки (DLL). Среда выполнения может поддерживать другие механизмы реализации внешних методов.

Если внешний метод содержит атрибут DllImport, объявление метода также должно включать модификатор static. Этот пример демонстрирует использование модификатора extern и атрибута DllImport.

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);  
}

### Разделяемые методы

Если объявление метода содержит модификатор partial, метод является разделяемым методом. Разделяемые методы могут объявляться только в качестве членов разделяемых типов (§10.2). На использование таких методов накладывается ряд ограничений. Дополнительные сведения о разделяемых методах см. в §10.2.7.

### Методы расширения

Если первый параметр метода содержит модификатор this, такой метод называется методом расширения. Методы расширения могут быть объявлены только в статических классах, не являющихся универсальными или вложенными. Первый параметр метода расширения не может содержать отличных от this модификаторов или иметь тип указателя.

В следующем примере показан статический класс, в котором объявляются два метода расширения.

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

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

Метод расширения представляет собой регулярный статический метод. Кроме того, в области действия включающего его статистического класса метод расширения может быть вызван с использованием синтаксиса вызова метода экземпляра (§7.6.5.2). В этом случае в качестве первого аргумента используется выражение-получатель.

В следующей программе используются объявленные выше методы расширения.

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in strings.Slice(1, 2)) {  
 Console.WriteLine(s.ToInt32());  
 }  
 }  
}

Метод Slice доступен для string[], а метод ToInt32 — для string, поскольку оба они объявлены как методы расширения. Эта программа аналогична следующей, в которой используются вызовы обычных статических методов.

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));  
 }  
 }  
}

### Тело метода

Тело метода содержит блок или точку с запятой.

Объявления абстрактных и внешних методов не предоставляют реализацию метода, поэтому их тела содержат только точку с запятой. Тело любого другого метода содержит блок (§8.2), включающий в себя операторы, которые выполняются при вызове метода.

Метод имеет тип результата void, если типом возвращаемого значения является void или, в случае асинхронного метода, типом возвращаемого значения является System.Threading.Tasks.Task. В противном случае типом результата синхронного метода является его тип возвращаемого значения, а типом результата асинхронного метода с типом возвращаемого значения System.Threading.Tasks.Task<*T*> — *T*.

Если типом результата метода является void, операторам return (§8.9.4) в теле этого метода не разрешено задавать выражение. Если выполнение метода, возвращающего void, завершается нормально (управление передается из конечной точки тела метода), этот метод просто возвращается его текущему вызывающему объекту.

Если тип результата метода отличен от void, каждый оператор return в теле этого метода должен задавать выражение, неявно преобразуемое к типу результата. Конечная точка тела метода, возвращающего значение, должна быть недостижима. Другими словами, в методе, возвращающем значение, не допускается передача управления из конечной точки тела метода.

В этом примере

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;  
 }  
 }  
}

метод F, возвращающий значение, порождает ошибку времени компиляции, поскольку в этом случае управление может быть передано из конечной точки тела метода. Методы G и H являются правильными, поскольку в них все возможные ветви выполнения заканчиваются оператором return, задающим возвращаемое значение.

### Перегрузка метода

Правила разрешения перегрузки метода описываются в разделе §7.5.2.

## Свойства

Свойство — это член, предоставляющий доступ к характеристикам объекта или класса. Примеры свойств: размер шрифта, заголовок окна, имя клиента и т. д. Свойства — это естественное расширение полей. Как свойства, так и поля являются именованными членами со связанными типами, для обращения к которым используется одинаковый синтаксис. Однако в отличие от полей, свойства не указывают места хранения. Вместо этого свойства содержат методы доступа, определяющие операторы, которые используются при чтении или записи их значений. Таким образом, свойства предоставляют механизмы, позволяющие связать определенные действия с чтением или записью атрибутов объекта. Кроме того, свойства обеспечивают вычисление таких атрибутов.

Свойства объявляются с помощью объявлений свойств:

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

Объявление свойства может включать набор атрибутов (§17) допустимое сочетание любых из четырех модификаторов доступа (§10.3.5), а также модификаторы new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) и extern (§10.6.7).

В отношении использования сочетаний модификаторов объявления свойств подчиняются тем же правилам, что и объявления методов (§10.6).

Тип свойства задает представленный в объявлении тип свойства. Имя члена задает имя свойства. Если в свойстве не содержится явная реализация члена интерфейса, имя члена представляет собой просто идентификатор. При явной реализации члена интерфейса (§13.4.1) имя члена состоит из типа интерфейса, точки "." и идентификатора (в указанной последовательности).

Тип свойства должен быть доступен, по меньшей мере, в такой же степени, как и само свойство (§3.5.4).

Объявления методов доступа заключаются в фигурные скобки "{" и "}" и объявляют методы доступа (§10.7.2) свойства. Методы доступа задают исполняемые операторы, связанные с чтением или записью свойства.

Несмотря на то что для обращения к свойствам и полям используется одинаковый синтаксис, свойство не классифицируется как переменная. Поэтому невозможно передать свойство как аргумент ref или out.

Если объявление свойства включает модификатор extern, свойство называется внешним свойством. Поскольку объявление внешнего свойства не предоставляет фактической реализации, каждое из его объявлений методов доступа состоит из точки с запятой.

### Статические свойства и свойства экземпляра

Если объявление свойства включает модификатор static, свойство называется статическим свойством. Если модификатор static отсутствует, свойство называется свойством экземпляра.

Статическое свойство не связано с конкретным экземпляром. При использовании зарезервированного слова this в методах доступа статического свойства возникает ошибка времени компиляции.

Свойство экземпляра связано с заданным экземпляром класса, обращение к которому может выполняться с помощью зарезервированного слова this (§7.6.7) в методах доступа свойства.

При доступе к члену (§7.6.4) с использованием ссылки на свойство в форме E.M для статического свойства M параметр E должен обозначать тип, содержащий M, а для свойства экземпляра M E должен обозначать экземпляр типа, содержащего M.

Различия между статическими членами и членами экземпляров рассматриваются в разделе §10.3.7.

### Методы доступа

Объявления методов доступа свойства задают исполняемые операторы, связанные с чтением или записью свойства.

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  
;

Объявление метода доступа включает в себя объявление метода доступа get, объявление метода доступа set или оба этих объявления. Каждое объявление метода доступа состоит из маркера get или set, за которым следуют необязательный модификатор метода доступа и тело метода доступа.

На использование модификаторов методов доступа налагаются следующие ограничения.

* Модификатор метода доступа не может использоваться в интерфейсе или явной реализации члена интерфейса.
* Для свойства или индексатора, не имеющего модификатора override, модификатор метода доступа может использоваться только в том случае, если свойство или индексатор содержит оба метода доступа (get и set), и применяется только к одному из них.
* Для свойства или индексатора, содержащего модификатор override, метод доступа должен соответствовать используемому модификатору метода доступа переопределяемого метода доступа.
* Модификатор метода доступа должен объявлять более строгий уровень доступа, чем уровень доступа самого свойства или индексатора. Более точно:
* Если для свойства или индексатора объявлен уровень доступа public, можно использовать модификаторы метода доступа с уровнем доступа protected internal, internal, protected и private.
* Если для свойства или индексатора объявлен уровень доступа protected internal, можно использовать модификаторы метода доступа с уровнем доступа internal, protected и private.
* Если для свойства или индексатора объявлен уровень доступа internal или protected, можно использовать модификатор метода доступа с уровнем доступа private.
* Если для свойства или индексатора объявлен уровень доступа private, использовать модификаторы метода доступа нельзя.

Для свойств с модификатором abstract или extern тело метода доступа содержит только точку с запятой. Свойства, не являющиеся абстрактными или внешними, могут быть автоматически реализуемыми свойствами. В этом случае задаются оба метода доступа (get и set), в теле которых содержится только точка с запятой (§10.7.3). Для методов доступа всех других свойств, не являющихся абстрактными или внешними, тело метода доступа является блоком, который задает операторы, исполняемые при вызове соответствующего метода доступа.

Метод доступа get соответствует не содержащему параметров методу, возвращаемое значение которого имеет тип свойства. За исключением случаев, когда свойство является конечным объектом операции присваивания, при ссылке на свойство в выражении вызывается метод доступа get для вычисления значения свойства (§7.1.1). Тело метода доступа get должно соответствовать правилам для возвращающих значение методов, описанным в §10.6.10. В частности, все операторы return в теле метода доступа get должны задавать выражение, которое может быть неявно преобразовано к типу свойства. Более того, конечная точка метода доступа get не должна достигаться.

Метод доступа set соответствует методу с типом возвращаемого значения void и одним параметром значения, имеющим тип свойства. Неявный параметр метода доступа set всегда имеет имя value. Если свойство является конечным объектом операции присваивания (§7.17) или операндом операторов ++ или -- (§7.6.9, §7.7.5), метод доступа set вызывается с аргументом (значение аргумента располагается в правой части операции присваивания или является операндом операторов ++ или --), который предоставляет новое значение (§7.17.1). Тело метода доступа set должно соответствовать правилам для возвращающих void методов, описанным в §10.6.10. В частности, операторам return в теле метода доступа set не разрешается задавать выражение. Объявление локальной переменной или константы в методе доступа set, имеющее имя value, является ошибкой времени компиляции, поскольку метод доступа set неявно имеет параметр с таким именем.

В зависимости от наличия или отсутствия методов доступа get и set свойства классифицируются следующим образом.

* Свойство, содержащее оба метода доступа (get и set), называется свойством для чтения и записи.
* Свойство, содержащее только метод доступа get, называется свойством только для чтения. Адресатом назначения является ошибка во время компиляции для свойства только для чтения.
* Свойство, содержащее только метод доступа set, называется свойством только для записи. Свойство, доступное только на запись, может использоваться в выражениях только в качестве конечного объекта операции присваивания. Во всех остальных случаях возникает ошибка времени компиляции.

В этом примере

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  
 }  
}

элемент управления Button объявляет открытое свойство Caption. Метод доступа get свойства Caption возвращает строку, хранящуюся в закрытом поле caption. Метод доступа set сравнивает новое значение с текущим, и если оно отличается, сохраняет новое значение и обновляет элемент управления. Для свойств часто используется описанный выше шаблон. Метод доступа get просто возвращает значение, хранящееся в закрытом поле, а метод set изменяет закрытое поле и выполняет определенные действия, необходимые для полного обновления состояния объекта.

В следующем примере показано использование свойства Caption для описанного выше класса Button.

Button okButton = new Button();  
okButton.Caption = "OK"; // Invokes set accessor  
string s = okButton.Caption; // Invokes get accessor

В этом примере метод доступа set вызывается посредством присваивания значения свойству, а метод доступа get — посредством ссылки на свойство в выражении.

Методы доступа get и set свойства не являются отдельными членами и не могут объявляться отдельно от свойства. Два метода доступа на чтение и запись не могут иметь различные уровни доступа. Пример:

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; }  
 }  
}

В этом примере не объявляется одно свойство, доступное на чтение и запись. Вместо этого в нем объявляется два одноименных свойства, одно из которых является свойством, доступным только на чтение, а другое — свойством, доступным только на запись. Поскольку два члена, объявленные в одном классе, не могут иметь одинаковые имена, возникает ошибка времени компиляции.

Если имя объявляемого в производном классе свойства совпадает с именем унаследованного свойства, свойство производного класса скрывает унаследованное свойство по отношению к операциям чтения и записи. В этом примере

class A  
{  
 public int P {  
 set {...}  
 }  
}

class B: A  
{  
 new public int P {  
 get {...}  
 }  
}

свойство P класса B скрывает свойство P класса A по отношению к операциям чтения и записи. Таким образом, в операторах

B b = new B();  
b.P = 1; // Error, B.P is read-only  
((A)b).P = 1; // Ok, reference to A.P

присваивание свойству b.P вызывает ошибку времени компиляции, поскольку свойство только для чтения P класса B скрывает свойство только для записи P класса A. Обратите внимание, что для доступа к скрытому свойству P можно использовать приведение.

В отличие от открытых полей свойства обеспечивают разделение между внутренним состоянием объекта и его открытым интерфейсом. Рассмотрим пример.

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; }  
 }  
}

В этом примере в классе Label используется два поля типа int — x и y — для хранения местоположения надписи. Сведения о местоположении предоставляются открытым образом в виде свойств X и Y, а также свойства Location типа Point. Если в последующих версиях класса Label потребуется внутреннее хранение данных о местоположении с помощью типа Point, это можно реализовать, не внося изменения в открытый интерфейс класса:

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; }  
 }  
}

Если бы свойства x и y были полями с модификатором public readonly, внести такие изменения в класс Label было бы невозможно.

Предоставление состояния с помощью свойств не обязательно является менее эффективным, чем непосредственное предоставление полей. В частности, если невиртуальное свойство содержит небольшой объем кода, в среде выполнения вызовы его методов доступа могут быть заменены фактическим кодом методов доступа. Этот процесс называется встраиванием и обеспечивает эффективность обращения к свойству на уровне обращения к полю, сохраняя при этом повышенную гибкость свойств.

Поскольку вызов метода доступа get концептуально равнозначен чтению значения поля, не рекомендуется, чтобы при использовании методов доступа get выполнялись видимые побочные действия. В этом примере

class Counter  
{  
 private int next;

public int Next {  
 get { return next++; }  
 }  
}

В этом примере значение свойства Next зависит от числа выполненных обращений к нему. Таким образом, обращение к свойству порождает заметные побочные действия, поэтому в этом случае свойство необходимо реализовывать как метод.

Соглашение об отсутствии побочных действий для методов доступа get не означает, что методы доступа get всегда должны использоваться только для возвращения значений, хранящихся в полях В действительности методы доступа get часто используются для вычисления значения свойства посредством обращения к нескольким полям или вызова методов. Однако правильно разработанный метод доступа get не выполняет действий, вызывающих заметные изменения в состоянии объекта.

Свойства можно использовать для задержки инициализации ресурса до того момента времени, когда на него будет сделана первая ссылка. Пример:

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;  
 }  
 }  
}

Класс Console содержит три свойства (In, Out и Error), которые представляют стандартные устройства ввода, вывода и вывода ошибок соответственно. Благодаря предоставлению этих членов в виде свойств класса Console обеспечивается задержка их инициализации до момента фактического использования Например, при первой ссылке на свойство Out, как показано в примере

Console.Out.WriteLine("hello, world");

создается базовый объект TextWriter для устройства вывода. Однако если приложение не ссылается на свойства In и Error, объекты для этих устройств не создаются.

### Автоматически реализуемые свойства

Для свойства, определенного как автоматически реализуемое свойство, автоматически создается скрытое резервное поле, для которого реализуются методы доступа для чтения и записи.

Следующий пример:

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

равнозначен следующему объявлению:

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; } }  
}

Поскольку резервное поле недоступно, чтение и запись его значений осуществляются только с помощью методов доступа свойства, даже внутри типа контейнера. Это означает, что автоматически реализуемые свойства только для чтения или только для записи не имеют смысла и не разрешены. Однако возможно задать различные уровни доступа для каждого из методов доступа. Таким образом, можно имитировать свойство только для чтения с закрытым резервным полем следующим образом:

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; }  
}

Это ограничение также означает, что явное присваивание типов структуры автоматически реализуемым свойствам может осуществляться только с помощью стандартного конструктора структуры, поскольку для присваивания самого свойства требуется явное присваивание структуры. Это означает, что пользовательские конструкторы должны вызывать конструкторы по умолчанию.

### Специальные возможности

Если метод доступа содержит модификатор метода доступа, область доступности (§3.5.2) метода определяется на основании объявленной доступности модификатора метода доступа. Если метод доступа не содержит модификатор метода доступа, область доступности метода определяется на основании объявленной доступности свойства или индексатора.

Наличие модификатора метода доступа никогда не влияет на поиск членов (§7.3) или разрешение перегрузки (§7.5.3). Модификаторы свойства или индексатора всегда определяют связанные свойство или индексатор независимо от контекста доступа.

После выбора конкретного свойства или индексатора допустимость их использования определяется на основании областей доступности задействованных методов доступа.

* Для использования в качестве значения (§7.1.1) должен существовать и быть доступным метод доступа get.
* Для использования в качестве конечного объекта операции простого присваивания (§7.17.1) должен существовать и быть доступным метод доступа set.
* Для использования в качестве конечного объекта операции составного присваивания (§7.17.2) или в качестве конечного объекта операторов ++ или -- (§7.5.9, §7.6.5) должны существовать и быть доступны оба метода доступа (get и set).

В следующем примере свойство A.Text скрывается свойством B.Text даже в тех контекстах, в которых вызывается только метод доступа set. Свойство B.Count, напротив, недоступно для класса M, поэтому вместо него используется доступное свойство A.Count.

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  
 }  
}

Метод доступа, используемый для реализации интерфейса, может не содержать модификатор метод доступа. Если для реализации интерфейса используется только один метод доступа, второй метод доступа может быть объявлен с помощью модификатора метода доступа:

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  
 }  
}

### Виртуальные, запечатанные, переопределяющие и абстрактные методы доступа

Объявление свойства virtual указывает, что методы доступа свойства являются виртуальными. Модификатор virtual применяется к обоим методам доступа свойства для чтения и записи. Объявление в качестве виртуального только одного метода доступа такого свойства не допускается

Объявление свойства abstract указывает, что методы доступа свойства являются виртуальными, однако не предоставляет их фактической реализации. Вместо этого необходимо определить неабстрактные производные классы, предоставляющие собственные реализации методов доступа посредством переопределения свойства. Поскольку метод доступа для объявления абстрактного свойства не предоставляет фактической реализации, тело метода доступа содержит только точку с запятой.

Объявление свойства, содержащее одновременно модификаторы abstract и override, указывает, что свойство является абстрактным и переопределяет базовое свойство. Методы доступа такого свойства также являются абстрактными.

Объявления абстрактных свойств допускаются только в абстрактных классах (§10.1.1.1). Методы доступа унаследованного виртуального свойства можно переопределить в производном классе, включив объявление свойства, задающее директиву override. Этот способ называется переопределяющим объявлением свойства. Переопределяющее объявление свойства не объявляет новое свойство. Вместо этого оно уточняет реализацию методов доступа существующего виртуального свойства.

Переопределяющее объявление свойства должно задавать в точности такие же модификаторы доступа, тип и имя, что и у унаследованного свойства. Если унаследованное свойство содержит только один метод доступа (то есть является свойством, доступным только для чтения или только для записи), переопределяющее свойство должно включать только этот метод доступа. Если унаследованное свойство содержит оба метода доступа (является свойством, доступным на чтение и запись), переопределяющее свойство может включать один или оба метода доступа.

Объявление переопределяющего свойства может содержать модификатор sealed. Использование этого модификатора позволяет предотвратить последующее переопределение свойства в производных классах. Методы доступа запечатанного свойства также являются запечатанными.

За исключением различий в синтаксисе объявления и вызова, поведение виртуальных, запечатанных, переопределяющих или абстрактных методов доступа в точности соответствует поведению виртуальных, запечатанных, переопределяющих или абстрактных методов. В частности, правила, описанные в разделах §10.6.3, §10.6.4, §10.6.5 и §10.6.6, применяются, как если бы методы доступа были методами соответствующего вида.

* Метод доступа get соответствует не имеющему параметров методу, имеющему возвращаемое значение типа свойства и те же модификаторы, что и содержащее свойство.
* Метод доступа set соответствует методу с типом возвращаемого значения void, одним параметром значения, имеющим тип свойства, и имеющему те же модификаторы, что и содержащее свойство.

В этом примере

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 — это виртуально свойство только для чтения, Y — это виртуальное свойство только для записи, а Z — это абстрактное свойство для чтения и записи. Поскольку свойство Z является абстрактным, содержащий его класс A также должен быть объявлен как абстрактный.

В следующем примере показан класс, производный от класса A.

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; }  
 }  
}

В этом примере объявления свойств X, Y и Z представляют собой переопределяющие объявления свойств. Каждое объявление свойства точно соответствует модификаторам доступа, типу и имени соответствующего наследуемого свойства. Метод доступа get свойства X и метод доступа set свойства Y используют ключевое слово base для обращения к наследуемым методам доступа. Объявление свойства Z переопределяет оба абстрактных метода доступа. Таким образом, в классе B отсутствуют необработанные члены абстрактной функции, в связи с чем класс B может быть неабстрактным классом.

Если свойство объявлено с помощью модификатора override, все переопределенные методы доступа должны быть доступны переопределяющему коду. Кроме того, объявленные уровни доступа для самого свойства или индексатора и его методов доступа должны соответствовать уровням доступа переопределенного члена и его методов доступа. Пример:

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  
 }  
}

## События

Событие — это член, используемый классом или объектом для уведомления. Клиенты могут присоединять к событиям исполняемый код с помощью обработчиков событий.

События объявляются с помощью объявлений событий:

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

Объявление события может включать набор атрибутов (§17) допустимое сочетание любых из четырех модификаторов доступа (§10.3.5), а также модификаторы new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) и extern (§10.6.7).

В отношении использования сочетаний модификаторов объявления событий подчиняются тем же правилам, что и объявления методов (§10.6).

Тип объявления события должен быть типом делегата (§4.2), и этот тип делегата должен иметь уровень доступности не меньший, чем у самого события (§3.5.4).

Объявление события может включать объявления методов доступа к событиям. Однако если такие объявления отсутствуют, они автоматически предоставляются компилятором для событий, не являющихся абстрактными или внешними (§10.8.1). Для внешних событий методы доступа предоставляются извне.

Объявление события, в котором опущены объявления методов доступа к событиям, определяет одно или несколько событий — по одному для каждого декларатора переменной. Атрибуты и модификаторы применяются ко всем членам, объявленным с помощью такого объявления события.

Если объявление события содержит одновременно модификатор abstract и заключенные в фигурные скобки объявления методов доступа событий, возникает ошибка времени компиляции.

Если объявление события содержит модификатор extern, событие является внешним событием. Поскольку объявления внешних событий не предоставляют фактической реализации, при наличии одновременно модификатора extern и объявлений методов доступа событий возникает ошибка.

Если декларатор переменной объявления события с модификатором abstract или external содержит инициализатор переменной, возникает ошибка времени компиляции.

Событие может использоваться в качестве левого операнда в операторах += и -= (§7.17.3). Эти операторы используются для присоединения к событию обработчика событий или его удаления соответственно. Модификаторы доступа события определяют контексты, в которых такие операции разрешены.

Операторы += и -= являются единственно допустимыми операторами для события вне типа, в котором оно объявлено. В связи с этим внешний код может добавлять обработчики к событию и удалять их, однако не может получать или изменять базовый список обработчиков событий.

В операциях вида x += y или x -= y (где для события x выполняется ссылка за пределы типа, который содержит объявление x) результат операции имеет тип void (а не значение x со значением x после присваивания). Это правило не допускает непосредственного просмотра базового делегата события внешним кодом.

В следующем примере показан порядок вложения обработчиков событий в экземпляры класса Button:

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  
 }  
}

В этом примере с помощью конструктора экземпляра LoginDialog создается два экземпляра класса Button и присоединяются обработчики событий Click.

### События, подобные полям

В тексте программы в классе или структуре, содержащих объявление события, некоторые события можно использовать как поля. Для использования таким образом событие не должно иметь модификаторы abstract и extern; кроме того, оно не может включать явные объявления методов доступа к событиям. Такое событие можно использовать в любом контексте, где разрешено использование полей. Поле содержит делегат (§15), ссылающийся на список обработчиков событий, добавленных к событию. Если обработчики событий не добавлены, поле содержит null.

В этом примере

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;  
 }  
}

Событие Click используется как поле внутри класса Button. Как показано в примере, это поле можно проверять, изменять и использовать в выражениях вызова делегата. Метод OnClick класса Button "вызывает" событие Click. Понятие вызова события совершенно эквивалентно вызову делегата, представленного событием. Поэтому не существует специальных языковых конструкций для вызова событий. Обратите внимание, что вызову делегата предшествует проверка, что делегат не равен null.

Вне объявления класса Button член Click может использоваться только с левой стороны операторов += и –=, как в следующем примере.

b.Click += new EventHandler(…);

В этом примере делегат добавляется к списку вызовов события Click, а в примере

b.Click –= new EventHandler(…);

делегат удаляется из списка вызовов события Click.

При компиляции события, подобного полю, компилятор автоматически создает хранилище для хранения делегата и создает методы доступа для события, добавляющие или удаляющие обработчики событий поля делегата. Операции добавления и удаления являются потокобезопасными; их можно (но не обязательно) выполнять при блокировке (§8.12) содержащего объекта для события экземпляра или объекта типа (§7.6.10.6) для статического события.

Так, объявление события экземпляра вида:

class X  
{  
 public event D Ev;  
}

будет откомпилировано в нечто, эквивалентное:

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 \*/  
 }  
 }  
}

В классе X ссылки на Ev on в левом операнде операторов += и –= приводят к вызову методов доступа add и remove. Все остальные ссылки на Ev компилируются в ссылки на скрытое поле \_\_Ev (§7.6.4). Имя "\_\_Ev" произвольное; скрытое поле могло бы иметь любое имя или вообще не иметь никакого имени.

### Методы доступа к событиям

В объявлениях событий обычно опускаются объявления доступа к событиям, как в вышеприведенном примере Button. В число причин для этого входит случай, когда затраты на хранение одного поля на каждое событие не являются приемлемыми. В таких случаях класс может включать объявления методов доступа к событиям и использовать частный механизм для хранения списка обработчиков событий.

Объявления методов доступа к событиям события указывают исполняемые операторы, связанные с добавлением и удалением обработчиков событий.

Объявления методов доступа состоят из объявления метода доступа add и объявления метода доступа remove. Каждое объявление метода доступа состоит из лексемы add или remove, за которой следует блок. В блоке, связанном с объявлением метода доступа add, указываются операторы, выполняемые при добавлении обработчика событий, а в блоке, связанном с объявлением метода доступа remove, указываются операторы, выполняемые при удалении обработчика событий.

Каждое объявление метода доступа add и объявление метода доступа remove соответствует методу с одним параметром значения типа события и типом возвращаемого значения void. Неявный параметр метода доступа к событию называется value. Когда событие используется в назначении события, используется соответствующий метод доступа к событию. В частности, если оператором присваивания является +=, используется метод доступа add, а если оператор присваивания -=, используется метод доступа remove. В любом случае правый операнд оператора присваивания используется в качестве аргумента метода доступа к событию. Блок объявления метода доступа add или объявления метода доступа remove должен соответствовать правилам для методов void, описанным в §10.6.10. В частности, операторам return в таком блоке запрещено указывать выражение.

Поскольку в методе доступа к событию неявно имеется параметр с именем value, объявление в методе доступа к событию локальной переменной или константы с таким именем является ошибкой времени компиляции.

В этом примере

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);  
 }  
}

класс Control реализует механизм внутреннего хранилища для событий. Метод AddEventHandler связывает значение делегата с ключом, метод GetEventHandler возвращает делегат, в данный момент связанный с ключом, а метод RemoveEventHandler удаляет делегат в качестве обработчика событий для указанного события. Предположительно, лежащий в основе механизм хранилища разработан так, что отсутствуют затраты на связывание значения делегата null с ключом, и таким образом необрабатываемые события не расходуют емкость хранилища.

### Статические события и события экземпляров

Если в объявление события включен модификатор static, событие называют статическим событием. Если нет модификатора static, событие называется событием экземпляра.

Статическое событие не связано с конкретным экземпляром, и обращение к this в методах доступа статического события является ошибкой времени компиляции.

Событие экземпляра связано с данным экземпляром класса, и к этому экземпляру можно обращаться как к this (§7.6.7) в методах доступа этого события.

Когда к событию обращаются через доступ к члену (§7.6.4) вида E.M, если M является статическим событием, E должно означать тип, содержащий M, а если M является событием экземпляра, E должно означать экземпляр типа, содержащего M.

Различия между статическими членами и членами экземпляров рассматриваются в разделе §10.3.7.

### Виртуальные, запечатанные, переопределяющие и абстрактные методы доступа

Объявление события как virtual указывает, что методы доступа этого события являются виртуальными. Модификатор virtual применяется к обоим методам доступа события.

Объявление события как abstract указывает, что методы доступа события являются виртуальными, но не предоставляет фактическую реализацию методов доступа. Вместо этого требуются не абстрактные производные классы для предоставления их собственной реализации для методов доступа посредством переопределения события. Поскольку объявление абстрактного события не предоставляет фактическую реализацию, оно не может предоставить разделенные скобками объявления методов доступа к событиям.

Объявление события, включающее оба модификатора abstract и override, указывает, что событие является абстрактным и переопределяет основное событие. Методы доступа такого события также являются абстрактными.

Объявления абстрактных событий разрешены только в абстрактных классах (§10.1.1.1).

Методы доступа унаследованного виртуального события могут быть переопределены в производном классе включением объявления события, указывающего модификатор override. Это называется объявлением переопределяющего события. Объявление переопределяющего события не объявляет новое событие. Вместо этого оно просто специализирует реализации методов доступа существующего виртуального события.

Объявление переопределяющего события должно указывать точно те же модификаторы доступа, тип и имя, что и переопределяемое событие.

Объявление переопределяющего события может включать модификатор sealed. Использование этого модификатора предотвращает дальнейшее переопределение события производным классом. Методы доступа запечатанного события также запечатаны.

Включение модификатора new в объявление переопределяющего события является ошибкой времени компиляции.

За исключением различий в синтаксисе объявления и вызова, поведение виртуальных, запечатанных, переопределяющих или абстрактных методов доступа в точности соответствует поведению виртуальных, запечатанных, переопределяющих или абстрактных методов. В частности, правила, описанные в разделах §10.6.3, §10.6.4, §10.6.5 и §10.6.6, применяются, как если бы методы доступа были методами соответствующего вида. Каждый метод доступа соответствует методу с единственным параметром-значением типа события, с типом возвращаемого значения void и с теми же модификаторами, что содержащее событие.

## Индексаторы

Индексатор является членом, дающим возможность индексировать объект так же, как массив. Индексаторы объявляются с помощью объявлений индексаторов:

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 ]

Объявление индексатора может включать набор атрибутов (§17) допустимое сочетание любых из четырех модификаторов доступа (§10.3.5), а также модификаторы new (§10.3.4), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) и extern (§10.6.7).

Объявления индексаторов подчиняются тем же правилам, что и объявления методов (§10.6), в отношении допустимых сочетаний модификаторов, с одним исключением – модификатор static не допускается в объявлении индексатора.

Модификаторы virtual, override и abstract являются взаимоисключающими, кроме одного случая. Модификаторы abstract и override могут использоваться совместно, так чтобы абстрактный индексатор мог переопределить виртуальный.

Тип в объявлении индексатора указывает тип элемента индексатора, вводимого объявлением. Если только индексатор не является явной реализацией члена интерфейса, за типом следует зарезервированное слово this. Для явной реализации члена интерфейса за типом следует тип интерфейса, «.» и зарезервированное слово this. В отличие от других членов, индексаторы не имеют имен, определяемых пользователем.

Список формальных параметров указывает параметры индексатора. Список формальных параметров индексатора соответствует списку формальных параметров метода (§10.6.1), кроме того, что по крайней мере один параметр должен быть указан, а модификаторы параметров ref и out не разрешены.

Тип индексатора и каждый из типов, на которые есть ссылки в списке формальных параметров, должны быть по крайней мере так же доступными, как сам индексатор (§3.5.4).

Объявления методов доступа (§10.7.2), которые должны быть заключены в лексемы "{" и "}", объявляют методы доступа индексатора. Методы доступа указывают исполняемые операторы, связанные с чтением и записью элементов индексатора.

Хотя синтаксис доступа к элементу индексатора такой же, как синтаксис доступа к элементу массива, элемент индексатора не классифицируется как переменная. Так, невозможно передать элемент индексатора в качестве аргумента ref или out.

Список формальных параметров индексатора определяет подпись (§3.6) индексатора. В частности, подпись индексатора состоит из числа и типов его формальных параметров. Тип элемента и имена формальных параметров не являются частью сигнатуры индексатора.

Сигнатура индексатора должна отличаться от сигнатур всех других индексаторов, объявленных в этом же классе.

Понятия индексаторов и свойств очень схожи, но отличаются следующим:

* свойство идентифицируется своим именем, а индексатор идентифицируется своей подписью;
* доступ к свойству производится через простое имя (§7.6.2) или доступ к члену (§7.6.4), а доступ к элементу индексатора производится через доступ к элементу (§7.6.6.2);
* свойство может быть членом static, а индексатор всегда является членом экземпляра;
* метод доступа get свойства соответствует методу без параметров, тогда как метод доступа get индексатора соответствует методу с тем же списком формальных параметров, что у индексатора;
* метод доступа set свойства соответствует методу с одним параметром с именем value, тогда как метод доступа set индексатора соответствует методу с тем же списком формальных параметров, что у индексатора, плюс дополнительный параметр с именем value;
* объявление в методе доступа к индексатору локальной переменной с тем же именем, что и параметр индексатора, является ошибкой времени компиляции;
* в объявлении переопределяющего свойства при обращении к унаследованному свойству используется синтаксис base.P, где P – имя свойства. В объявлении переопределяющего индексатора при обращении к унаследованному индексатору используется синтаксис base[E], где E – список выражений, разделенных запятыми.

За исключением этих различий, все правила, определенные в §10.7.2 и §10.7.3, применяются к методам доступа к индексатору, а также к методам доступа к свойствам.

Если в объявление индексатора включен модификатор extern, индексатор называется внешним индексатором. Так как объявление внешнего индексатора не предоставляет фактической реализации, каждое из его объявлений метода доступа состоит из точки с запятой.

В примере внизу объявлен класс BitArray, реализующий индексатор для доступа к отдельным битам в битовом массиве.

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);  
 }  
 }  
 }  
}

Экземпляр класса BitArray расходует существенно меньше памяти, чем соответствующий bool[] (так как каждое значение первого занимает только один бит, тогда как у второго – один байт), но позволяет выполнять те же операции, что и bool[].

В следующем классе CountPrimes используется BitArray и классический алгоритм «решето» для вычисления количества простых чисел между 1 и заданным максимумом.

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);  
 }  
}

Обратите внимание, что синтаксис обращения к элементам BitArray точно такой же, как для bool[].

В следующем примере показан класс сетки 26 × 10 с индексатором с двумя параметрами. Первым параметром должна быть буква верхнего или нижнего регистра в диапазоне A–Z, а вторым – целое число в диапазоне 0–9.

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;  
 }  
 }  
}

### Перегрузка индексатора

Правила разрешения перегрузки индексаторов описаны в §7.5.2.

## Операторы

Оператор является членом, определяющим значение оператора выражения, который можно применить к экземплярам класса. Операторы объявляются с помощью объявлений операторов:

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  
;

Есть три категории перегружаемых операторов: унарные операторы (§10.10.1), бинарные операторы (§10.10.2) и операторы преобразования (§10.10.3).

Если в объявление оператора включен модификатор extern, оператор называется внешним оператором. Так как внешний оператор не предоставляет фактическую реализацию, его тело оператора состоит из точки с запятой. Для всех других операторов тело оператора состоит из блока, в котором указаны операторы, выполняемые при вызове этого оператора. Блок оператора должен удовлетворять правилам для методов, возвращающих значение, описанным в §10.6.10.

Следующие правила применяются ко всем объявлениям операторов:

* объявление оператора должно включать оба модификатора: public и static;
* параметрами оператора должны быть параметры-значения (§5.1.4). Указание в объявлении оператора параметров ref или out является ошибкой времени компиляции;
* сигнатура оператора (§10.10.1, §10.10.2, §10.10.3) должна отличаться от сигнатур всех других операторов, объявленных в этом же классе;
* все типы, на которые ссылается объявление оператора, должны быть, по крайней мере, так же доступными, как сам оператор (§3.5.4);
* неоднократное появление одного и того же модификатора в объявлении оператора является ошибкой времени компиляции.

Каждая категория операторов налагает дополнительные ограничения, как описано в следующих разделах.

Подобно другим членам, операторы, объявленные в базовом классе, наследуются производными классами. Так как объявления операторов всегда требуют участия класса или структуры, где объявлен оператор, в сигнатуре оператора, то для оператора, объявленного в производном классе, невозможно скрыть оператор, объявленный в базовом классе. Таким образом, модификатор new никогда не требуется и поэтому никогда не допускается в объявлении оператора.

Дополнительные сведения об унарных и бинарных операторах можно найти в §7.3.

Дополнительные сведения об операторах преобразования можно найти в §6.4.

### Унарные операторы.

К объявлениям унарных операторов применяются следующие правила, здесь T обозначает тип экземпляра класса или структуры, где содержится объявление оператора:

* унарный оператор +, -, ! или ~ должен принимать единственный параметр типа T или T? и может возвращать любой тип;
* унарный оператор ++ или -- должен принимать единственный параметр типа T или T? и должен возвращать тот же самый тип или тип, производный от него;
* унарный оператор true или false должен принимать единственный параметр типа T или T? и возвращать значение типа bool.

Подпись унарного оператора состоит из лексемы оператора (+, -, !, ~, ++, --, true или false) и типа единственного формального параметра. Тип возвращаемого значения не является частью сигнатуры унарного оператора, как и имя формального параметра.

Для унарных операторов true и false требуется попарное объявление. Если в классе объявлен один из этих операторов без объявления другого, возникает ошибка времени компиляции. Операторы true и false описаны далее в разделах §7.12.2 и §7.20.

В следующем примере показана реализация и последующее применение operator ++ для класса целочисленного вектора.

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  
 }  
}

Обратите внимание, как метод оператора возвращает значение, созданное добавлением 1 к операнду, точно так же, как операторы постфиксного увеличения и уменьшения (§7.6.9) и операторы префиксного увеличения и уменьшения (§7.7.5). В отличие от C++, этому методу не требуется непосредственно изменять значение своего операнда. Фактически изменение значения операнда нарушала бы стандартную семантику оператора постфиксного увеличения.

### Бинарные операторы

К объявлениям бинарных операторов применяются следующие правила, здесь T обозначает тип экземпляра класса или структуры, где содержится объявление оператора:

* бинарный оператор не-сдвига должен принимать два параметра, по крайней мере один из которых должен иметь тип T или T?, и может возвращать любой тип;
* бинарный оператор << или >> должен принимать два параметра, первый из которых должен иметь тип T или T?, а второй должен иметь тип int или int? и может возвращать любой тип.

Подпись бинарного оператора состоит из лексемы оператора (+, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >= или <=) и типов этих двух формальных параметров. Тип возвращаемого значения и имена формальных параметров не являются частью сигнатуры бинарного оператора.

Для некоторых бинарных операторов требуется попарное объявление. Для каждого объявления одного из операторов пары должно быть соответствующее объявление другого оператора из пары. Два объявления операторов соответствуют, если у них одинаковый тип возвращаемого значения и одинаковый тип каждого параметра. Для следующих операторов требуется попарное объявление:

* operator == и operator !=
* operator > и operator <
* operator >= и operator <=

### Операторы преобразования

Объявление оператора преобразования вводит пользовательское преобразование (§6.4), которое дополняет предопределенные неявные и явные преобразования.

Объявление оператора преобразования, включающее зарезервированное слово implicit, вводит неявное преобразование, определенное пользователем. Неявные преобразования могут происходить в разнообразных ситуациях, включая вызовы членов функций, приведение выражений и присваивания. Это описано далее в §6.1.

Объявление оператора преобразования, включающее зарезервированное слово explicit, вводит явное преобразование, определенное пользователем. Явные преобразования могут происходить в выражениях приведения, они описаны далее в §6.2.

Оператор преобразования выполняет преобразование из исходного типа, указанного типом параметра оператора преобразования, в конечный тип, указанный возвращаемым типом оператора преобразования.

Для данного исходного типа S и конечного типа T, если S или T является типом, допускающим присваивание пустой ссылки, пусть S0 и T0 ссылаются на свои основные типы, иначе S0 и T0 равны S и T соответственно. В классе или структуре разрешено объявлять преобразование от исходного типа S к конечному типу T, если только справедливо все следующее:

* S0 и T0 являются разными типами;
* либо S0, либо T0 является типом структуры или класса, где имеет место объявление этого оператора;
* ни S0, ни T0 не является типом интерфейса;
* без преобразований, определенных пользователем, не существует преобразование от S к T или от T к S.

Для выполнения этих правил любые параметры типа, связанные с S или T, считаются уникальными типами, не имеющими отношений наследования с другими типами, а любые ограничения на эти параметры типа игнорируются.

В этом примере

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  
}

первые два объявления операторов разрешены, так как, по замыслу §10.9.3, T, int и string соответственно считаются уникальными типами без отношений. Однако третий оператор ошибочен, так как C<T> является базовым классом для D<T>.

Из второго правила следует, что оператор преобразования должен преобразовывать либо в направлении к типу класса или структуры, либо в направлении от типа класса или структуры, где оператор объявлен. Например, можно для типа класса или структуры C определить преобразование из C в int и из int в C, но не из int в bool.

Невозможно непосредственно переопределить предопределенное преобразование. Так, операторам преобразования запрещено преобразовывать от или к object, так как уже существуют неявные и явные преобразования между object и всеми другими типами Аналогично, ни исходный, ни конечный тип преобразования не может быть базовым типом другого, так как в этом случае преобразование уже существует.

Однако возможно объявить операторы для универсальных типов, которые для аргументов особого типа указывают преобразования, уже существующие как предопределенные. В этом примере

struct Convertible<T>  
{  
 public static implicit operator Convertible<T>(T value) {...}

public static explicit operator T(Convertible<T> value) {...}  
}

если тип object указан в качестве аргумента типа для T, второй оператор объявляет преобразование, которое уже существует (неявное и, следовательно, также и явное преобразование существует от любого типа к типу object).

В тех случаях, когда существует предопределенное преобразование между двумя типами, любые пользовательские преобразования между этими типами игнорируются. А именно:

* если существует неявное предопределенное преобразование (§6.1) от типа S к типу T, все определенные пользователем преобразования (неявные или явные) от S к T игнорируются;
* если существует явное предопределенное преобразование (§6.2) от типа S к типу T, любые пользовательские явные преобразования от S к T игнорируются. Более того:
* если T является типом интерфейса, пользовательские неявные преобразования от S к T игнорируются.
* В противном случае пользовательские неявные преобразования от S к T все же признаются.

Для всех типов, кроме object, операторы, объявленные типом Convertible<T> вверху, не противоречат предопределенным преобразованиям. Пример:

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  
}

Однако для типа object предопределенные преобразования скрывают пользовательские преобразования во всех случаях, кроме одного:

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  
}

Пользовательским преобразованиям запрещено преобразование из типов интерфейса или в типы интерфейса. В частности, это ограничение обеспечивает отсутствие каких-либо пользовательских преобразований при преобразовании в тип интерфейса; кроме того, оно гарантирует, что преобразование в тип интерфейса будет успешно выполнено, только если преобразуемый объект действительно реализует указанный тип интерфейса.

Сигнатура оператора преобразования состоит из исходного типа и конечного типа. (Обратите внимание, что это единственный вид члена, в котором тип возвращаемого значения участвует в сигнатуре). Классификация implicit или explicit оператора преобразования не является частью подписи оператора. Так, в классе или структуре нельзя объявить операторы преобразования и implicit, и explicit с теми же исходными и конечными типами.

В общем, пользовательские неявные преобразования следует разрабатывать так, чтобы они никогда не вызывали исключения и не теряли информацию. Если определенное пользователем преобразование может вызывать исключения (например, потому что исходный аргумент вне допустимого диапазона) или терять информацию (например, отбрасывать старшие разряды), то такое преобразование следует определить как явное преобразование.

В этом примере

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);  
 }  
}

В этом примере преобразование от Digit к byte неявное, так как оно никогда не вызывает исключения и не теряет информацию, а преобразование от byte к Digit явное, так как Digit может представлять только поднабор возможных значений byte.

## Конструкторы экземпляров

Конструктор экземпляра является членом, реализующим действия, необходимые для инициализации экземпляра класса. Конструкторы экземпляров объявляются с помощью объявлений конструкторов:

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  
;

Объявление конструктора может включать набор атрибутов (§17), допустимое сочетание из четырех модификаторов доступа (§10.3.5) и модификатор extern (§10.6.7). В объявление конструктора запрещено включать один и тот же модификатор несколько раз.

Идентификатор в деклараторе конструктора должен указывать имя класса, в котором объявлен конструктор экземпляров. Если указано любое другое имя, происходит ошибка времени компиляции.

Необязательный список формальных параметров конструктора экземпляров подчиняется тем же правилам, что и список формальных параметров метода (§10.6). Список формальных параметров определяет подпись (§3.6) конструктора экземпляров и управляет процессом, посредством которого разрешение перегрузки (§7.5.2) выбирает отдельный конструктор экземпляров в вызове.

Каждый тип, на который есть ссылка в списке формальных параметров конструктора экземпляров, должен быть, по крайней мере, так же доступным, как сам конструктор (§3.5.4).

Необязательный инициализатор конструктора указывает другой конструктор экземпляров для вызова перед выполнением операторов, заданных в теле конструктора этого конструктора экземпляров. Это описано далее в §10.11.1.

Если в объявление конструктора включен модификатор extern, конструктор называется внешним конструктором. Поскольку объявление внешнего конструктора не предоставляет фактическую реализацию, его тело конструктора состоит из точки с запятой. Во всех других конструкторах тело конструктора состоит из блока, в котором указаны операторы для инициализации нового экземпляра класса. Это в точности соответствует блоку экземпляра метода с типом возвращаемого значения void (§10.6.10).

Конструкторы экземпляров не наследуются. Таким образом, у класса нет конструкторов экземпляров, отличных от тех, что действительно объявлены в этом классе. Если класс не содержит объявления конструкторов экземпляров, автоматически предоставляется конструктор экземпляров по умолчанию (§10.11.4).

Конструкторы экземпляров вызываются выражениями создания объектов (§7.6.10.1) и посредством инициализаторов конструкторов.

### Инициализаторы конструкторов

Все конструкторы экземпляров (за исключением конструкторов для класса object) неявно включают вызов другого конструктора экземпляров непосредственно перед телом конструктора. Конструктор, который должен быть неявно вызван, определяется инициализатором конструктора:

* инициализатор конструктора экземпляров вида base(список аргументовopt) побуждает вызов конструктора экземпляров из прямого базового класса. Этот конструктор выбирается с помощью списка аргументов и правил разрешения перегрузки в §7.5.3. Набор конструкторов экземпляров – кандидатов включает все доступные конструкторы экземпляров, содержащиеся в прямом базовом классе, или конструктор по умолчанию (§10.11.4), если в прямом базовом классе не объявлены конструкторы экземпляров. Если этот набор пуст или если не удается идентифицировать один наилучший конструктор экземпляров, возникает ошибка времени компиляции.
* инициализатор конструктора экземпляров вида this(список аргументовopt) побуждает вызов конструктора экземпляров из самого класса. Конструктор выбирается с помощью списка аргументов и правил разрешения перегрузки в §7.5.3. Набор конструкторов экземпляров – кандидатов состоит из всех доступных конструкторов экземпляров, объявленных в самом классе. Если этот набор пуст или если не удается идентифицировать один наилучший конструктор экземпляров, возникает ошибка времени компиляции. Если объявление конструктора экземпляров включает инициализатор конструктора, вызывающий сам конструктор, возникает ошибка времени компиляции.

Если у конструктора экземпляров нет инициализатора конструктора, неявно предоставляется инициализатор конструктора вида base(). Таким образом, объявление конструктора экземпляров вида

C(...) {...}

в точности эквивалентно

C(...): base() {...}

Область видимости параметров, заданных списком формальных параметров объявления конструктора экземпляров, включает инициализатор конструктора этого объявления. Таким образом, инициализатору конструктора разрешен доступ к параметрам конструктора. Пример:

class A  
{  
 public A(int x, int y) {}  
}

class B: A  
{  
 public B(int x, int y): base(x + y, x - y) {}  
}

Инициализатор конструктора экземпляров не может иметь доступ к создаваемому экземпляру. Поэтому ссылка на this в выражении аргумента инициализатора конструктора, как и ссылка в выражении аргумента на любой член экземпляра посредством простого имени, является ошибкой времени компиляции.

### Инициализаторы переменных экземпляров

Если у конструктора экземпляра нет инициализатора конструктора или есть инициализатор конструктора вида base(...), этот конструктор неявно выполняет инициализации, указанные инициализаторами переменных полей экземпляра, объявленных в его классе. Это соответствует последовательности присваиваний, выполняемых непосредственно после входа в конструктор и перед неявным вызовом конструктора прямого базового класса. Инициализаторы переменных выполняются в том текстовом порядке, в каком они представлены в объявлении класса.

### Выполнение конструктора

Инициализаторы переменных преобразуются в операторы присваивания, и эти операторы присваивания выполняются перед вызовом конструктора экземпляров базового класса. Этот порядок обеспечивает инициализацию всех полей экземпляра их инициализаторами переменных до выполнения каких-либо операторов, имеющих доступ к этому экземпляру.

Пример.

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);  
 }  
}

В этом примере, если new B() используется для создания экземпляра B, создается следующий вывод:

x = 1, y = 0

Значение x равно 1, потому что инициализатор переменной выполняется до вызова конструктора экземпляров базового класса. Однако значение y равно 0 (значение по умолчанию для int), потому что присваивание для y выполняется только после возврата из конструктора базового класса.

Полезно считать инициализаторы переменных экземпляра операторами, которые автоматически вставляются перед телом конструктора. Пример:

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;  
 }  
}

содержится несколько инициализаторов переменных, а также содержатся инициализаторы конструкторов обоих видов (base и this). Этот пример соответствует коду, показанному ниже, где каждый комментарий указывает автоматически вставляемый оператор (синтаксис, используемый для автоматически вставляемых вызовов конструктора не является допустимым, а служит только для пояснения механизма).

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) {  
 B(100); // Invoke B(int) constructor  
 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;  
 }  
}

### Конструкторы по умолчанию

Если класс не содержит объявления конструкторов экземпляров, автоматически предоставляется конструктор экземпляров по умолчанию. Этот конструктор по умолчанию просто вызывает не имеющий параметров конструктор прямого базового класса. Если класс абстрактный, то объявленная доступность для конструктора по умолчанию защищена. Иначе объявленная доступность для конструктора по умолчанию является общей. Таким образом, конструктор по умолчанию всегда имеет вид

protected C(): base() {}

или

public C(): base() {}

где C – это имя класса. Если при разрешении перегрузки не удается определить уникальный наиболее подходящий инициализатор конструктора базового класса, во время компиляции возникают ошибки.

В этом примере

class Message  
{  
 object sender;  
 string text;  
}

В этом примере предоставлен конструктор по умолчанию, так как класс не содержит объявлений конструкторов экземпляров. Так, этот пример в точности эквивалентен следующему:

class Message  
{  
 object sender;  
 string text;

public Message(): base() {}  
}

### Закрытые конструкторы

Если в классе T объявлены только закрытые конструкторы экземпляров, для классов вне программного текста T невозможно вывести из T или непосредственно создать экземпляры T. Поэтому, если класс содержит только статические члены и не предназначен для создания экземпляров, добавление пустого закрытого конструктора экземпляров предотвратит создание экземпляров. Пример:

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) {...}  
}

Класс Trig группирует связанные методы и константы, но не предназначен для создания экземпляров. Поэтому он объявляет единственный пустой закрытый конструктор экземпляров. По крайней мере один конструктор экземпляров должен быть объявлен, чтобы запретить автоматическое создание конструктора по умолчанию.

### Необязательные параметры конструктора экземпляров

Вид this(...) инициализатора конструктора обычно используется в сочетании с перегрузкой для реализации необязательных параметров конструктора экземпляров. В этом примере

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  
 }  
}

В этом примере первые два конструктора экземпляров только предоставляют значения по умолчанию для отсутствующих аргументов. Оба используют инициализатор конструктора this(...) для вызова третьего конструктора экземпляров, который действительно выполняет работу по инициализации нового экземпляра. Это действие соответствует действию необязательных параметров конструктора:

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");

## Статические конструкторы

Статический конструктор – это член, реализующий действия, необходимые для инициализации закрытого типа класса. Статические конструкторы объявляются с помощью объявлений статического конструктора:

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

Объявление статического конструктора может включать набор атрибутов (§17) и модификатор extern (§10.6.7).

Идентификатор в объявлении статического конструктора должен указывать имя класса, в котором объявлен статический конструктор. Если указано любое другое имя, происходит ошибка времени компиляции.

Если в объявление статического конструктора включен модификатор extern, статический конструктор называется внешним статическим конструктором. Так как объявление внешнего статического конструктора не предоставляет фактическую реализацию, его тело статического конструктора состоит из точки с запятой. Для всех других объявлений статического конструктора тело статического конструктора состоит из блока, в котором указаны операторы, которые необходимо выполнить, чтобы инициализировать класс. Это в точности соответствует телу метода статического метода с типом возвращаемого значения void (§10.6.10).

Статические конструкторы не наследуются и их нельзя вызвать непосредственно.

Статические конструкторы для закрытого типа класса выполняются не более одного раза в данной области приложения. Выполнение статического конструктора запускается первым из следующих событий, происходящих внутри области приложения:

* создан экземпляр типа класса;
* возникла ссылка на любой статический член типа класса.

Если класс содержит метод Main (§3.1), в котором начинается исполнение, статический конструктор для этого класса выполняется до вызова метода Main.

Для инициализации нового закрытого типа класса сначала создается новый набор статических полей (§10.5.1) для этого особого типа класса. Каждое из статических полей инициализируется своим значением по умолчанию (§5.2). Затем выполняются инициализаторы этих статических полей (§10.4.5.1). Наконец выполняется статический конструктор.

Пример:

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");  
 }  
}

Выходом этого примера должно быть:

Init A  
A.F  
Init B  
B.F

так как выполнение статического конструктора класса A запускается по обращению к A.F, а выполнение статического конструктора класса B запускается обращением к B.F.

Можно создать циклические зависимости, позволяющие наблюдать статические поля с инициализаторами переменных в их состоянии значений по умолчанию.

Пример:

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);  
 }  
}

производятся следующие выходные данные

X = 1, Y = 2

Для выполнения метода Main система сначала выполняет инициализатор для B.Y, до статического конструктора класса B. Инициализатор Y вызывает запуск статического конструктора класса A, так как имеется ссылка на значение A.X. Статический конструктор A в свою очередь продолжает вычислять значение X и при этом вычислении выбирает значение по умолчанию для Y, которое равно нулю. A.XA.X таким образом инициализируется значением 1. Процесс выполнения для класса A инициализаторов статических полей и статического конструктора затем завершается, возвращаясь к вычислению начального значения Y, результат которого становится равным 2.

Так как статический конструктор выполняется ровно один раз для каждого закрытого сформированного типа класса, он является удобным местом для принудительных проверок во время выполнения параметров-типов, которые не удается проверить с помощью ограничений во время компиляции (§10.1.5). Например, следующий тип использует статический конструктор для принудительной установки того, что аргументом типа является перечисляемый тип:

class Gen<T> where T: struct  
{  
 static Gen() {  
 if (!typeof(T).IsEnum) {  
 throw new ArgumentException("T must be an enum");  
 }  
 }  
}

## Деструкторы

Деструктор является членом, реализующим действия, необходимые для уничтожения экземпляра класса. Деструктор объявляется с помощью объявления деструктора:

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

Объявление деструктора может включать набор атрибутов (§17).

Идентификатор в деклараторе деструктора должен указывать имя класса, в котором объявлен деструктор. Если указано любое другое имя, происходит ошибка времени компиляции.

Если в объявление деструктора включен модификатор extern, деструктор называется внешним деструктором. Поскольку объявление внешнего деструктора не предоставляет фактическую реализацию, его тело деструктора состоит из точки с запятой. Во всех других деструкторах тело деструктора состоит из блока, в котором указаны операторы для уничтожения экземпляра класса. Тело деструктора в точности соответствует телу метода метода экземпляра с типом возвращаемого значения void (§10.6.10).

Деструкторы не наследуются. Таким образом, класс не имеет других деструкторов, кроме того, который может быть объявлен в этом классе.

Так как деструктор должен не иметь параметров, его нельзя перегрузить, поэтому класс может иметь не более одного деструктора.

Деструкторы вызываются автоматически, их нельзя вызвать явно. Экземпляр становится пригодным для уничтожения, когда более невозможно использование этого экземпляра любым кодом. Выполнение деструктора для экземпляра может произойти в любое время после того, как экземпляр становится пригодным для уничтожения. Когда экземпляр уничтожен, вызываются деструкторы в цепочке наследования этого экземпляра по порядку, от старшего производного к младшему. Деструктор может выполняться в любом потоке. Дальнейшее обсуждение правил, определяющих, когда и как выполняется деструктор, см. в §3.9.

Выполнение примера

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();  
 }  
}

даст на выходе:

B’s destructor  
A’s destructor

так как деструкторы в цепочке наследования вызываются по порядку, от самого старшего производного до самого младшего.

Деструкторы реализуются переопределением виртуального метода Finalize для System.Object. Программам C# запрещено переопределять этот метод или вызывать его (или его переопределения) непосредственно. Например, программа

class A   
{  
 override protected void Finalize() {} // error

public void F() {  
 this.Finalize(); // error  
 }  
}

содержит две ошибки.

Поведение компилятора такое, как будто этот метод и его переопределения вовсе не существуют. Таким образом, эта программа:

class A   
{  
 void Finalize() {} // permitted  
}

является допустимой, а показанный метод скрывает метод Finalize объекта System.Object.

Обсуждение поведения, когда исключение вызывается из деструктора, см. в §16.3.

## Итераторы

Член функции (§7.5), реализованный с помощью блока итератора(§8.2), называется итератором.

Блок итератора можно использовать как тело члена функции, пока тип возвращаемого значения соответствующего члена функции является одним из интерфейсов перечислителя (§10.14.1) или одним из перечислимых интерфейсов (§10.14.2). Он может встречаться как тело метода, тело оператора или тело метода доступа, тогда как события, конструкторы экземпляров, статические конструкторы и деструкторы не могут быть реализованы как итераторы.

Если член функции реализован с помощью блока итератора, указание любого из параметров ref или out в списке формальных параметров члена функции является ошибкой времени компиляции.

### Интерфейсы перечислителя

Интерфейсы перечислителя – это неуниверсальный интерфейс System.Collections.IEnumerator и все экземпляры универсального интерфейса System.Collections.Generic.IEnumerator<T>. Эти интерфейсы в этой главе для краткости обозначаем как IEnumerator и IEnumerator<T> соответственно.

### Перечислимые интерфейсы

Перечислимые интерфейсы – это неуниверсальный интерфейс System.Collections.IEnumerable и все экземпляры универсального интерфейса System.Collections.Generic.IEnumerable<T>. Эти интерфейсы в этой главе для краткости обозначаем как IEnumerable и IEnumerable<T> соответственно.

### Тип yield

Итератор создает последовательность значений одного и того же типа. Этот тип называется типом yield итератора.

* Типом выдачи итератора, который возвращает интерфейс IEnumerator или IEnumerable, является object.
* Типом выдачи итератора, который возвращает интерфейс IEnumerator<T> или IEnumerable<T>, является T.

### Объекты перечислителя

Если член функции, возвращающий тип интерфейса перечислителя, реализован с помощью блока итератора, при вызове члена функции не происходит немедленное выполнение кода в блоке итератора. Вместо этого создается и возвращается объект перечислителя. Этот объект инкапсулирует код, указанный в блоке итератора, а выполнение кода в блоке итератора происходит тогда, когда вызывается метод MoveNext объекта перечислителя. Объект перечислителя имеет следующие характеристики:

* Он реализует IEnumerator и IEnumerator<T>, где T – это тип yield итератора.
* Он реализует System.IDisposable.
* Он инициализируется копией значений аргумента (при их наличии) и значением экземпляра, переданным члену функции.
* У него есть четыре потенциальных состояния: before, running, suspended и after, а также начальное состояние before.

Обычно объект перечислителя является экземпляром созданного компилятором класса перечислителя, который инкапсулирует код в блоке итератора и реализует интерфейсы перечислителя, но возможны и другие методы реализации. Если класс перечислителя создан компилятором, этот класс будет вложен, прямо или косвенно, в класс, содержащий член функции, у него будет частная доступность и имя, зарезервированное для использования компилятором (§2.4.2).

Объект перечислителя может реализовать больше интерфейсов, чем указано выше.

В следующих разделах описано точное поведение членов MoveNext, Current и Dispose реализаций интерфейсов IEnumerable и IEnumerable<T>, предоставляемых объектом перечислителя.

Обратите внимание, что объекты перечислителя не поддерживают метод IEnumerator.Reset. Вызов этого метода приводит к исключению System.NotSupportedException.

#### Метод MoveNext

Метод MoveNext объекта перечислителя инкапсулирует код блока итератора. При вызове метода MoveNext выполняется код в блоке итератора и устанавливается соответствующее свойство Current объекта перечислителя. Точное действие, выполняемое MoveNext, зависит от состояния объекта перечислителя при вызове MoveNext:

* Если объект перечислителя находится в состоянии before, то при вызове MoveNext:
* состояние меняется на running;
* инициализируются параметры (включая this) блока итератора значениями аргументов и значением экземпляра, сохраненными при инициализации объекта перечислителя;
* выполняется блок итератора от начала и до тех пор, пока выполнение не будет прервано (как описано ниже).
* Если состояние объекта перечислителя running, результат вызова MoveNext не определен.
* Если состояние объекта перечислителя suspended, при вызове MoveNext:
* состояние меняется на running;
* восстанавливаются значения всех локальных переменных и параметров (включая this) к значениям, сохраненным при последней приостановке выполнения блока итератора. Обратите внимание, что содержимое любых объектов, на которые ссылаются эти переменные, могло измениться со времени предыдущего вызова MoveNext;
* возобновляется выполнение блока итератора с оператора, непосредственно следующего за оператором yield return, вызвавшим приостановку выполнения, и продолжается до тех пор, пока выполнение не будет прервано (как описано ниже).
* Если состояние объекта перечислителя after, при вызове MoveNext возвращается false.

Когда MoveNext выполняет блок итератора, выполнение может быть прервано четырьмя способами: оператором yield return, оператором yield break, по концу блока итератора и инициированным исключением, распространенным из блока итератора.

* При обнаружении оператора yield return (§8.14):
* выражение, заданное в операторе, вычисляется, неявно преобразуется к типу yield и присваивается свойству Current объекта перечислителя;
* выполнение тела итератора приостанавливается. Значения всех локальных переменных и параметров (включая this) сохраняются, как и место этого оператора yield return. Если оператор yield return находится внутри одного или нескольких блоков try, соответствующие им блоки finally не выполняются в это время.
* состояние объекта перечислителя меняется на suspended;
* метод MoveNext возвращает true вызвавшему его, указывая этим, что итерация успешно продвинулась к следующему значению.
* При обнаружении оператора yield break (§8.14):
* Если оператор yield break находится внутри одного или нескольких блоков try, соответствующие им блоки finally выполняются;
* состояние объекта перечислителя меняется на after;
* метод MoveNext возвращает false вызвавшему его, указывая этим, что итерация выполнена.
* Если встретился конец тела итератора:
* состояние объекта перечислителя меняется на after;
* метод MoveNext возвращает false вызвавшему его, указывая этим, что итерация выполнена.
* Если инициируется исключение и распространяется из блока итератора:
* соответствующие блоки finally в теле итератора будут выполняться распространением исключения;
* состояние объекта перечислителя меняется на after;
* распространение исключения продолжается до вызвавшего метод MoveNext.

#### Свойство Current

На свойство Current объекта перечислителя воздействуют операторы yield return в блоке итератора.

Если объект перечислителя находится в состоянии suspended, значением Current является значение, установленное предыдущим вызовом MoveNext. Если объект перечислителя находится в состоянии before, running или after, результат обращения к Current не определен.

Для итератора с типом выдачи, отличным от object, результат обращения к Current посредством реализации IEnumerable объекта перечислителя соответствует обращению к Current посредством реализации IEnumerator<T> объекта перечислителя и приведением результата к object.

#### Метод Dispose

Метод Dispose используется для очистки итерации приведением объекта перечислителя в состояние after.

* Если состояние объекта перечислителя before, вызов Dispose меняет это состояние на after.
* Если состояние объекта перечислителя running, результат вызова Dispose не определен.
* Если состояние объекта перечислителя suspended, при вызове Dispose:
* состояние меняется на running;
* выполняются все блоки finally, как если бы последний выполненный оператор yield return был оператором yield break. Если это приводит к инициированию и распространению из тела итератора исключения, состояние объекта перечислителя устанавливается в after, а исключение распространяется до объекта, вызвавшего метод Dispose;
* состояние меняется на after.
* Если состояние объекта перечислителя after, вызов Dispose не оказывает влияние.

### Перечислимые объекты

Если член функции, возвращающий перечислимый тип интерфейса, реализован с помощью блока итератора, при вызове члена функции не происходит немедленное выполнение кода в блоке итератора. Вместо этого создается и возвращается перечислимый объект. Метод GetEnumerator перечислимого объекта возвращает объект перечислителя, который инкапсулирует код, указанный в блоке итератора, а выполнение кода в блоке итератора происходит при вызове метода MoveNext объекта перечислителя. Перечислимый объект имеет следующие характеристики:

* Он реализует IEnumerable и IEnumerable<T>, где T – это тип yield итератора.
* Он инициализируется копией значений аргумента (при их наличии) и значением экземпляра, переданным члену функции.

Обычно перечислимый объект является экземпляром созданного компилятором перечислимого класса, который инкапсулирует код в блоке итератора и реализует перечислимые интерфейсы, но возможны и другие методы реализации. Если перечислимый класс создан компилятором, этот класс будет вложен, прямо или косвенно, в класс, содержащий член функции, у него будет частная доступность и имя, зарезервированное для использования компилятором (§2.4.2).

Перечислимый объект может реализовать больше интерфейсов, чем указано выше. В частности, перечислимый объект может также реализовать IEnumerator и IEnumerator<T>, позволяя ему выступать в роли как перечислимого, так и перечислителя. При таком типе реализации при первом вызове метода GetEnumerator перечислимого объекта возвращается сам перечислимый объект. При последующих вызовах GetEnumerator для перечислимого объекта, при их наличии, будет возвращена копия перечислимого объекта. Таким образом, каждый возвращенный перечислитель имеет собственное состояние, и изменения в одном перечислителе не влияют на другой.

#### Метод GetEnumerator

Перечислимый объект предоставляет реализацию методов GetEnumerator интерфейсов IEnumerable и IEnumerable<T>. Два метода GetEnumerator совместно используют общую реализацию, которая получает и возвращает доступный объект перечислителя. Объект перечислителя инициализируется значениями аргументов и значением экземпляра, сохраненными при инициализации перечислимого объекта, но в других отношениях объект перечислителя функционирует так, как описано в §10.14.4.

### Пример реализации

В этом разделе описана возможная реализация итераторов при использования стандартных конструкций C#. В основе реализации, описанной в этом разделе, лежат принципы, используемые в компиляторе Microsoft C#, однако она никоим образом не является обязательной или единственной возможной.

Следующий класс Stack<T> реализует метод GetEnumerator с помощью итератора. Итератор перечисляет элементы стека в нисходящем порядке.

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];  
 }  
}

Метод GetEnumerator можно транслировать в экземпляр класса перечислителя, созданного компилятором и инкапсулирующего код в блоке итератора, как показано в следующем примере.

class Stack<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1: IEnumerator<T>, IEnumerator  
 {  
 int \_\_state;  
 T \_\_current;  
 Stack<T> \_\_this;  
 int i;

public \_\_Enumerator1(Stack<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.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 IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

В предыдущей трансляции код в блоке итератора преобразован в конечный автомат и помещен в метод MoveNext класса перечислителя. Более того, локальная переменная i преобразована в поле в объекте перечислителя, что делает возможным ее существование при последующих вызовах MoveNext.

Следующий пример служит для печати простой таблицы умножения целых чисел от 1 до 10. В этом примере метод FromTo возвращает перечислимый объект и реализуется с помощью итератора.

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.Write("{0,3} ", x \* y);  
 }  
 Console.WriteLine();  
 }  
 }  
}

Метод FromTo можно транслировать в экземпляр класса перечислителя, созданного компилятором и инкапсулирующего код в блоке итератора, как показано в следующем примере.

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();  
 }  
 }  
}

Перечислимый класс реализует как перечислимые интерфейсы, так и интерфейсы перечислителя, что позволяет ему служить как перечислимым классом, так и перечислителем. При первом вызове метода GetEnumerator будет возвращен сам перечислимый объект. При последующих вызовах GetEnumerator для перечислимого объекта, при их наличии, будет возвращена копия перечислимого объекта. Таким образом, каждый возвращенный перечислитель имеет собственное состояние, и изменения в одном перечислителе не влияют на другой. Метод Interlocked.CompareExchange используется для обеспечения потокобезопасной операции.

Параметры from и to преобразуются в поля в перечислимом классе. Поскольку параметр from изменяется в блоке итератора, то дополнительно вводится поле \_\_from, которое содержит начальное значение, присваиваемое параметру from в каждом перечислителе.

Метод MoveNext порождает исключение InvalidOperationException, если при его вызове значение \_\_state равно 0. Это позволяет не допустить использования перечислимого объекта в качестве объекта перечислителя без вызова GetEnumerator.

В следующем примере показан класс простого дерева. Класс Tree<T> реализует метод GetEnumerator с помощью итератора. Итератор перечисляет элементы дерева в инфиксном порядке.

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));  
 }

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.Write("{0} ", i);  
 Console.WriteLine();

Tree<string> strings = MakeTree(  
 "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");  
 foreach (string s in strings) Console.Write("{0} ", s);  
 Console.WriteLine();  
 }  
}

Метод GetEnumerator можно транслировать в экземпляр класса перечислителя, созданного компилятором и инкапсулирующего код в блоке итератора, как показано в следующем примере.

class Tree<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1 : IEnumerator<T>, IEnumerator  
 {  
 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;

\_\_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();  
 }  
 }  
}

Создаваемые компилятором временные данные, используемые в операторах foreach, переносятся в поля \_\_left и \_\_right объекта перечислителя. Поле \_\_state объекта перечислителя обновляется таким образом, чтобы при создании исключения был правильно вызван необходимый метод Dispose(). Обратите внимание, что написание транслируемого кода с простыми операторами foreach невозможно.

## Асинхронные функции

Асинхронной функцией называется метод (§10.6) или анонимная функция (§7.15) с модификатором async. В общем смысле, термин асинхронный используется для описания любого вида функции с модификатором async.

Если в списке формальных параметров асинхронной функции указаны параметры ref или out, возникает ошибка времени компиляции.

Типом возвращаемого значения асинхронного метода должен быть void или тип задачи. Типы задач — System.Threading.Tasks.Task и типы, сформированные из System.Threading.Tasks.Task<T>. Эти типы в этой главе для краткости обозначаем как Task и Task<T> соответственно. Асинхронный метод, возвращающий тип задачи, называется возвращающим задачу.

Точное определение типов задач зависит от конкретной реализации, но с точки зрения языка, тип задачи — это одно из следующих состояний: incomplete, succeeded или faulted. Задача faulted записывает соответствующее исключение. Успешно выполненная задача Task<*T*> записывает тип результата *T*. Типы задач поддерживают ожидание и поэтому могут быть операндами выражений await (§7.7.7).

Асинхронная функция может приостановить вычисление с помощью выражений await (§7.7.7), содержащихся в ее теле. Позднее вычисление может быть возобновлено с точки приостанавливающего выражения await посредством делегата возобновления. Делегат возобновления — это тип System.Action; при его вызове вычисление, выполняемое асинхронной функцией, возобновляется с выражения await, в котором произошла приостановка. Текущим вызывающим объектом асинхронной функции является исходный вызывающий объект, если выполнение функции не приостанавливалось, или последний вызывающий объект делегата возобновления, если приостановка имела место.

### Вычисление асинхронной функции, возвращающей задачу

Вызов асинхронной функции, возвращающей задачу, приводит к созданию экземпляра типа возвращаемой задачи. Этот экземпляр называется возвращаемой задачей асинхронной функции. Сначала задача находится в состоянии incomplete.

Затем вычисляется тело асинхронной функции до ее приостановки (по достижении выражения await) или прекращения, после чего управление вместе с возвращаемой задачей передается вызывающему объекту.

После прекращения выполнения тела асинхронной функции возвращаемая задача переходит в состояние incomplete:

* если выполнение тела функции прекращается в результате достижения оператора return или конца тела, результирующее значение записывается в возвращаемой задаче, которая переходит в состояние succeeded;
* если выполнение тела функции прекращается в результате неперехваченного исключения (§8.9.5), в возвращаемой задаче записывается исключение и задача переходит в состояние faulted.

### Вычисление асинхронной функции, возвращающей значение void

Если асинхронная функция имеет тип возвращаемого значения void, процесс вычисления отличается от описанного выше следующим образом. Поскольку задача не возвращается, функция сообщает о завершении и исключениях в контекст синхронизации текущего потока. Точное определение контекста синхронизации зависит от конкретной реализации, но этот контекст представляет "место", в котором выполняется поток. Контекст синхронизации получает уведомления при начале вычисления асинхронной функции, возвращающей значение void, при его успешном завершении, а также при возникновении неперехваченных исключений.

Это позволяет контексту синхронизации отслеживать количество выполняемых в нем асинхронных функций, возвращающих значение void, и принимать решения относительно обработки получаемых от этих функций исключений.

# Структуры

Структуры, как и классы, представляют определенным образом организованные данные. В структурах могут содержаться как данные, так и функции. Однако структуры, в отличие от классов, являются типами значений и для них не требуется выделение памяти в куче. Переменная с типом структуры непосредственно содержит данные этой структуры, в то время как переменная с типом класса содержит ссылку на данные, которые считаются объектом.

Структуры особенно удобны для работы с небольшим объемом данных, имеющих семантику значения. Примерами структур являются комплексные числа, точки в системе координат или словарные пары «ключ-значение». Ключевой особенностью таких структур данных является наличие нескольких переменных-членов, для которых не требуется использовать наследование или идентификацию на уровне ссылок и которые удобно реализовывать с использованием семантики значения, когда при присваивании копируется значение, а не ссылка.

Как описано в §4.1.4, доступные в C# простые типы, такие как int, double и bool, фактически являются структурами. Благодаря тому, что эти предопределенные типы являются структурами, в языке C# можно реализовывать новые «простые» типы при помощи переопределения структур и операторов. В конце этой главы приводится два примера таких типов (§11.4).

## Объявления структур

Объявление структуры является объявлением типа (§9.6), в котором объявляется новая структура:

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

Объявление структуры состоит из необязательного набора атрибутов (§17), за которым следует необязательный набор модификаторов структуры (§11.1.1), необязательный модификатор partial, ключевое слово struct, идентификатор, который указывает имя структуры, необязательная спецификация списка параметров типа (§10.1.3), необязательная спецификация интерфейсов структуры (§11.1.2) ), необязательная спецификация конструкций ограничений для параметров типа (§10.1.5), тело структуры (§11.1.4) и точка с запятой (необязательно).

### Модификаторы структуры

Объявление структуры может включать последовательность модификаторов структуры:

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

Включение одного модификатора в объявление структуры несколько раз приведет к возникновению ошибки времени выполнения.

Модификаторы объявления структуры имеют такое же действие, как модификаторы объявления класса (§10.1).

### Модификатор partial

Модификатор partial указывает, что объявление структуры является объявлением частичного типа. Несколько частичных объявлений структуры с одним именем в едином пространстве имен или объявлении типа объединяются в одно объявление структуры с соблюдением правил, указанных в §10.2.

### Интерфейсы структуры

Объявление структуры может содержать спецификацию интерфейсов структуры, в этом случае считается, что в структуре непосредственно реализованы заданные типы интерфейсов.

struct-interfaces:  
: interface-type-list

Реализация интерфейсов рассматривается более подробно в §13.4.

### Тело структуры

Тело структуры определяет членов структуры.

struct-body:  
{ struct-member-declarationsopt }

## Члены структуры

К членам структуры относятся члены, включенные в объявления членов структуры и члены, унаследованные из System.ValueType.

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

За исключением различий, рассмотренных в §11.3, описания членов класса, содержащиеся в §10.3–§10.14, справедливы и для членов структуры.

## Различия между классом и структурой

Структуры имеют несколько важных отличий от классов:

* Структуры являются типами значения (§11.3.1).
* Все типы структур неявным образом наследуются из класса System.ValueType (§11.3.2).
* При присваивании переменной с типом структуры создается копия присваиваемого значения (§11.3.3).
* Значением по умолчанию для структуры является значение, при котором все поля типов значения устанавливаются в соответствующие значения по умолчанию, а все поля ссылочного типа — в значение null (§11.3.4).
* Операции упаковки и распаковки используются для выполнения преобразования между типом структуры и объектом object (§11.3.5).
* Ключевое слово this в структурах имеет другой смысл (§7.6.7).
* Объявления полей экземпляров для структуры не могут включать инициализаторы переменных (§11.3.7).
* В структуре не разрешается объявлять конструктор экземпляров без параметров (§11.3.8).
* В структуре не может быть объявлен деструктор (§11.3.9).

### Семантика значений

Структуры относятся к типам значения (§4.1) и считаются имеющими семантику значения. Наоборот, классы относятся к ссылочным типам (§4.2) и считаются имеющими семантику ссылок.

Переменная с типом структуры непосредственно содержит данные этой структуры, в то время как переменная с типом класса содержит ссылку на данные, которые считаются объектом. Если структура B содержит поле экземпляра типа A и A является типом структуры, зависимость A от B или от типа, созданного на основе B, приведет к возникновению ошибки компиляции. Структура X имеет прямую зависимость от структуры Y в том случае, если X содержит поле экземпляра с типом Y. В соответствии с этим определением полный набор структур, по отношению к которым структура является зависимой, представляет собой транзитивное замыкание связи с типом имеет прямую зависимость от. Пример

struct Node  
{  
 int data;

Node next; // error, Node directly depends on itself

}

является ошибкой, поскольку структура Node содержит поле экземпляра с собственным типом. Другой пример. Код

struct A { B b; }

struct B { C c; }

struct C { A a; }

является ошибкой, поскольку типы A, B и C зависят друг от друга.

В классах две переменные могут ссылаться на один объект, в результате чего действия с одной переменной могут повлиять на объект, на который ссылается другая переменная. В структурах каждая из переменных имеет собственную копию данных (за исключением переменных параметров ref и out) и действия с одной из переменных не могут повлиять на другую переменную. Более того, поскольку структуры не относятся к ссылочным типам, значения с типом структуры не могут быть равны null.

При объявлении

struct Point  
{  
 public int x, y;

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

выполнение фрагмента кода

Point a = new Point(10, 10);  
Point b = a;  
a.x = 100;  
System.Console.WriteLine(b.x);

возвратит значение 10. При присваивании значения переменной a переменной b создается копия этого значения, поэтому на переменную b не влияет изменение значения переменной a.x. Если бы структура Point была объявлена как класс, выходным значением было бы 100, поскольку переменные a и b ссылались бы на один и тот же объект.

### Наследование

Все типы структур неявным образом наследуются от класса System.ValueType, который, в свою очередь, наследуется от класса object. В объявлении структуры может быть указан список реализованных интерфейсов, однако здесь не допускается указание базового класса.

Тип структуры никогда не является абстрактным и всегда неявным образом запечатан. Поэтому в объявлении структуры не допускаются модификаторы abstract и sealed.

Так как для структур не поддерживается наследование, член структуры не может иметь объявленный уровень доступа protected или protected internal.

Функции-члены в структуре не могут иметь модификаторы abstract или virtual, а модификатор override допускается только для переопределения методов, унаследованных от класса System.ValueType.

### Присваивание

При присваивании переменной с типом структуры создается копия присваиваемого значения. В отличие от этого при выполнении присваивания в переменную с типом класса копируется ссылка, а не сам объект, на который указывает эта ссылка.

Аналогичным образом при передаче структуры в качестве параметра по значению или возвращения ее в результате выполнения функции-члена создается копия структуры. Структура может передаваться в функцию-член по ссылке с использованием параметра ref или out.

Если целевым объектом операции присваивания является свойство или индексатор структуры, в качестве переменной должно быть указано выражение экземпляра, сопоставленное доступу к индексатору или свойству. Если выражение экземпляра классифицировано как значение, возникнет ошибка компиляции. Более подробно эта тема рассматривается в §7.17.1.

### Значения по умолчанию

Как рассматривалось в §5.2, некоторые виды переменных при создании автоматически инициализируются значениями по умолчанию. Для переменных с типом класса или другими ссылочными типами используется значение null. Однако в связи с тем, что структуры имеют тип значения и не могут быть равны null, значение по умолчанию для структуры является значением, полученным путем установки для всех полей с типом значения соответствующих значений по умолчанию, а для всех полей с ссылочным типом — значения null.

Для объявленной выше структуры Point строка

Point[] a = new Point[100];

выполняет инициализацию каждой переменной Point в массиве значением, полученным путем установки для полей x и y нулевых (0) значений.

Значение по умолчанию структуры соответствует значению, возвращаемому конструктором структуры по умолчанию (§4.1.2). Для структуры, в отличие от класса, не допускается объявление конструктора экземпляра без параметров. Вместо этого каждая структура неявно содержит конструктор экземпляра без параметров, который всегда возвращает значение, полученное путем установки для всех полей с типом значения соответствующих значений по умолчанию, а для всех полей с ссылочным типом — значения null.

Структуры должны создаваться таким образом, чтобы состояние инициализации по умолчанию было допустимым состоянием. В этом примере

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;  
 }  
}

пользовательский конструктор экземпляра обеспечивает защиту от пустых значений только при явном вызове. В случае, когда переменная KeyValuePair инициализируются значением по умолчанию, поля key и value имеют значение NULL, и в структуре следует предусмотреть обработку такого состояния.

### Упаковка и распаковка

Значение с типом класса можно преобразовать в тип object или в тип интерфейса, реализуемого этим классом, путем обработки данной ссылки во время компиляции как другого типа. Аналогичным образом значение с типом object или типом интерфейса можно преобразовать обратно в тип класса без изменения ссылки (естественно, в этом случае требуется проверка во время выполнения).

Так как структуры не относятся к ссылочным типам, для типов структуры эти операции реализуются по-другому. При преобразовании значения с типом структуры в тип object или в тип интерфейса, реализуемый этой структурой, происходит операция упаковки. Точно так же при обратном преобразовании значения с типом object или значения с типом интерфейса в тип структуры выполняется операция распаковки. Ключевое отличие от аналогичных операций с типами класса состоит в том, что при упаковке и распаковке выполняется копирование значения структуры в упакованный экземпляр или из такого экземпляра. Таким образом, после выполнения операции упаковки или распаковки изменения, внесенные в распакованную структуру, не отражаются в упакованной структуре.

Если тип структуры переопределяет виртуальный метод, унаследованный от класса System.Object (например, Equals, GetHashCode или ToString), вызов этого виртуального метода через экземпляр типа структуры не приводит к выполнению упаковки. Это правило действует даже в том случае, когда структура используется в качестве параметра типа и вызов происходит в экземпляре с типом параметра типа. Пример:

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>();  
 }  
}

Результат выполнения примера:

1  
2  
3

Несмотря на то, что использование метода ToString для выполнения побочных действий является плохим стилем, в этом примере демонстрируется, что при трех вызовах метода x.ToString() упаковка не выполнялась.

Аналогичным образом упаковка не выполняется неявным образом при доступе к члену с ограниченным параметром-типом. Например, интерфейс ICounter содержит метод Increment, который можно использовать для изменения значения. Если метод ICounter используется в качестве ограничения, реализация метода Increment вызывается со ссылкой на переменную, для которой был вызван метод Increment, а не для упакованной копии.

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>();  
 }  
}

При первом вызове метода Increment изменяется значение переменной x. Это не равноценно второму вызову метода Increment, при котором изменяется значение упакованной копии x. Таким образом, в результате выполнения программы будет получен следующий результат:

0  
1  
1

Дополнительные сведения об упаковке и распаковке см. в §4.3.

### Действие ключевого слова this

В конструкторе экземпляра или члене-функции экземпляра класса ключевое слово this классифицируется как значение. Таким образом, хотя ключевое слово this можно использовать для ссылки на экземпляр, для которого была вызвана данная функция-член, присваивание для this в функции-члене класса недоступно.

В конструкторе экземпляра структуры ключевое слово this соответствует параметру out типа структуры, а в функции-члене экземпляра структуры ключевое слово this соответствует параметру ref типа структуры. В обоих случаях this считается переменной, что позволяет изменить всю структуру, для которой была вызвана эта функция-член, путем присваивания в this или передачи this в качестве параметра ref или out.

### Инициализаторы полей

Как рассматривалось в §11.3.4, значение структуры по умолчанию состоит из значения, полученного путем установки для всех полей с типом значения соответствующих значений по умолчанию, а для всех полей ссылочного типа — значения null. По этой причине в структуре объявления полей экземпляра не могут содержаться инициализаторы переменных. Это ограничение действует только в отношении полей экземпляра. Статические поля структуры могут содержать инициализаторы переменных.

Пример:

struct Point  
{  
 public int x = 1; // Error, initializer not permitted  
 public int y = 1; // Error, initializer not permitted  
}

является неправильным, так как объявления полей экземпляра содержат инициализаторы переменных.

### Конструкторы

Для структуры, в отличие от класса, не допускается объявление конструктора экземпляра без параметров. Вместо этого каждая структура неявно содержит конструктор экземпляра без параметров, который всегда возвращает значение, полученное путем установки для всех полей с типом значения соответствующих значений по умолчанию, а для всех полей с ссылочным типом — значения NULL (§4.1.2). В структуре можно объявлять конструкторы экземпляров с параметрами. Пример

struct Point  
{  
 int x, y;

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

В приведенном выше объявлении операторы

Point p1 = new Point();

Point p2 = new Point(0, 0);

создают объекты Point, свойства которых x и y инициализированы нулевыми значениями.

Конструктор экземпляра структуры не может содержать инициализатор конструктора в форме base(...).

Если в конструкторе экземпляра структуры не указан инициализатор конструктора, переменная this соответствует параметру out типа структуры, при этом, аналогично параметру out, объект this должен быть определенно присвоен (§5.3) в каждой точке возвращения из конструктора. Если в конструкторе экземпляра структуры указан инициализатор конструктора, переменная this соответствует параметру ref с типом структуры и, аналогично параметру ref, переменная this считается определенно присвоенной в точке входа в тело конструктора. Рассмотрим приведенную ниже реализацию конструктора:

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  
 }  
}

Ни одна из функций-членов экземпляра (включая методы доступа set для свойств X и Y) не может быть вызвана до тех пор, пока все поля создаваемой структуры не будут определенно присвоены. Однако, если бы объект Point был классом, а не структурой, реализация конструктора экземпляра была бы разрешена.

### Деструкторы

В структуре не разрешается объявлять деструктор.

### Статические конструкторы

Для статических конструкторов структур в основном действуют те же правила, что и для статических конструкторов классов. Выполнение статического конструктора структуры запускается первым из следующих событий в домене приложения:

* Ссылка на статический член с типом структуры.
* Вызов явным образом объявленного конструктора с типом структуры.

Создание значений по умолчанию (§11.3.4) с типом структуры не ведет к вызову статического конструктора. В качестве примера можно указать начальные значения элементов массива.

## Примеры структур

Ниже приводится два примера использования типов struct для создания типов, которые могут использоваться аналогично предопределенным типам языка, но имеют измененную семантику.

### Тип целочисленного значения в базе данных

Представленная ниже структура DBInt реализует тип integer, который может представлять полный набор значений с типом int, а также дополнительное состояние, указывающее на неизвестное значение. Тип с такими характеристиками повсеместно используется в базах данных.

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”;  
 }  
}

### Логический тип базы данных

Представленная ниже структура DBBool реализует трехзначный логический тип. Возможные значения этого типа: DBBool.True, DBBool.False и DBBool.Null, где член Null указывает неизвестное значение. Подобные трехзначные логические типы часто используются в базах данных.

using System;

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";  
 }  
}

# Массивы

Массив представляет собой структуру данных, содержащую ряд переменных, доступ к которым осуществляется с использованием расчетных индексов. Все переменные, содержащиеся в массиве, которые также называются элементами массива, имеют одинаковый тип, который называется типом элементов массива.

Массив имеет ранг, определяющий количество индексов, связанных с каждым из элементов массива. Другими словами, ранг массива указывает количество его измерений. Массив с рангом, равным единице, называется одномерным массивом. Массив с рангом больше единицы называется многомерным массивом. Многомерные массивы конкретного размера часто называются двумерными, трехмерными и так далее.

Каждое измерение массива имеет сопоставленную ему длину, выраженную в виде целого неотрицательного числа. Длины измерений не являются частью типа массива, они устанавливаются при создании экземпляра массива с конкретным типом во время выполнения. Длина измерения определяет допустимый диапазон индексов для этого измерения: для измерения с длиной N индексы могут находиться в диапазоне от 0 до N – 1 включительно. Общее количество элементов в массиве равно произведению длин всех его размерностей. Если одно или несколько измерений массива имеют нулевую длину, массив считается пустым.

Элемент массива может иметь любой тип, в том числе тип массива.

## Типы массива

Тип массива записывается как тип не массива, за которым следуют спецификации ранга:

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 ,

Тип не массива является одним из типов, которые сами по себе не являются типом массива.

Ранг типа массива задается левой спецификацией ранга в типе массива: спецификация ранга указывает, что массив имеет ранг, равный единице плюс число лексем «,» в этой спецификации.

Тип элемента массива представляет собой тип, полученный в результате удаления крайней левой спецификации ранга:

* Тип массива в форме T[R] указывает массив с рангом R и типом элемента (типом не массива) T.
* Тип массива в виде T[R][R1]...[RN] указывает массив с рангом R и типом элементов T[R1]...[RN].

В результате данные спецификации ранга считываются слева направо перед последним типом элемента (типом не массива). Тип int[][,,][,] указывает одномерный массив трехмерных массивов из двумерных массивов значений с типом int.

Во время выполнения значение типа массива может быть равно null или может содержать ссылку на экземпляр массива этого типа.

### Тип System.Array

Тип System.Array является абстрактным базовым типом для всех типов массива. Выполняется неявное преобразование ссылок (§6.1.6) из любого типа в тип System.Array, а также явное преобразование ссылок (§6.2.4) из типа System.Array в любой тип массива. Обратите внимание, что тип System.Array сам по себе не является типом массива. Это тип класса, на основе которого создаются все типы массива.

Во время выполнения значение типа System.Array может быть равно null или являться ссылкой на экземпляр любого типа массива.

### Массивы и универсальный интерфейс IList

В одномерном массиве T[] реализуется интерфейс System.Collections.Generic.IList<T> (сокращенно IList<T>) и его базовые интерфейсы. В связи с этим выполняется неявное преобразование из T[] в IList<T> и его базовые интерфейсы. В дополнение к этому при наличии неявного преобразования ссылок из S в T в массиве S[] реализуется интерфейс IList<T> и выполняется неявное преобразование ссылок из S[] в IList<T> и его базовые интерфейсы (§6.1.6). При наличии явного преобразования ссылок из S в T выполняется явное преобразование ссылок из S[] в IList<T> и его базовые интерфейсы (§6.2.4). Пример:

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  
 }  
}

Присваивание lst2 = oa1 приведет к ошибке компилирования, поскольку преобразование из object[] в IList<string> должно быть явным и не может выполняться неявно. Приведение типов в строке (IList<string>)oa1 приведет к созданию исключения во время выполнения, так как переменная oa1 ссылается на объект object[], а не на объект string[]. Однако приведение типов в строке (IList<string>)oa2 приведет к созданию исключения во время выполнения, так как переменная oa2 ссылается на объект string[].

При выполнении явного или неявного преобразования ссылок из S[] в IList<T> также выполняется явное преобразование ссылок из интерфейса IList<T> и его базовых интерфейсов в S[] (§6.2.4).

Если в типе массива S[] реализуется интерфейс IList<T>, некоторые из членов реализованного интерфейса могут создавать исключения. Описание точного поведения этой реализации интерфейса выходит за рамки настоящей спецификации.

## Создание массива

Экземпляры массива создаются при помощи выражений\_создания\_массива (§7.6.10.4) либо путем объявлений полей или локальных переменных, содержащих инициализатор\_массива (§12.6).

Ранг массива и длина каждого из его измерений устанавливаются при создании экземпляра массива и остаются неизменными в течение всего времени жизни этого экземпляра. Другими словами, нельзя изменить ранг массива или длину его измерений.

Экземпляр массива всегда имеет тип массива. Тип System.Array является абстрактным типом, создание экземпляров которого невозможно.

Элементы массива, созданного с использованием выражений\_создания\_массива, всегда инициализируются значениями по умолчанию (§5.2).

## Доступ к элементам массива

Доступ к элементам массива осуществляется при помощи выражений доступа к элементам (§7.6.6.1) в форме A[I1, I2, ..., IN], где A является выражением типа массива, а каждый элемент IX является выражением типа int, uint, long, ulong либо может быть неявно преобразован в один или несколько из этих типов. Результатом осуществления доступа к элементу массива является переменная, а именно элемент массива, выбранный по индексу.

Элементы массива могут перечисляться с использованием оператора foreach (§8.8.4).

## Члены массива

Каждый тип массива наследуют члены, объявленные типом System.Array.

## Ковариация массивов

Если для любых двух переменных A и B ссылочного типа выполняется неявное (§6.1.6) или явное преобразование ссылок (§6.2.4) из A в B, такое же преобразование ссылок доступно из массива A[R] в массив B[R], где R указывает спецификацию\_ранга (одинаковую для обоих типов массивов). Эта связь называется ковариацией массивов. Ковариация массивов в частности означает, что значение с типом массива A[R] фактически может быть ссылкой на экземпляр типа массива B[R] при условии, что доступно неявное преобразование ссылок из B в A.

В связи с существованием ковариации массива при присваивании элементам массива ссылочного типа выполняется проверка времени выполнения, которая гарантирует, что присваиваемое элементу массива значение имеет допустимый тип (§7.17.1). Пример:

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);  
 }  
}

Присвоение массиву array[i] в методе Fill включает неявную проверку времени выполнения, гарантирующую, что объект, на который ссылается аргумент value, имеет значение null или является экземпляром типа, совместимого с фактическим типом элементов array. В методе Main первые два вызова метода Fill выполняются успешно, но третий вызов приводит к созданию исключения System.ArrayTypeMismatchException после выполнения первого присвоения массиву array[i]. Исключение возникает в связи с тем, что упакованное значение int не может храниться в массиве с типом string.

Ковариация массивов не расширяется до массивов типов значений. Например, не существует преобразования, разрешающего обрабатывать int[] как object[].

## Инициализаторы массива

Инициализаторы массива указываются в объявлениях полей (§10.5), объявлениях локальных переменных (§8.5.1) и выражениях создания массива (§7.6.10.4):

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

Инициализатор массива состоит из последовательности переменных инициализаторов, заключенных в фигурные скобки «{» и «}» и разделенных запятыми «,». Каждый инициализатор переменной представляет собой выражение или (для многомерных массивов) инициализатор вложенного массива.

Контекст использования инициализатора массива определяет тип инициализируемого массива. В выражении создания массива тип массива непосредственно предшествует инициализатору или получается из выражений в инициализаторе массива. В объявлении поля или переменной тип массива представляет собой тип объявляемого поля или переменной. Если инициализатор массива используется в объявлении поля или переменной, например

int[] a = {0, 2, 4, 6, 8};

он просто представляет собой краткую запись соответствующего выражения создания массива

int[] a = new int[] {0, 2, 4, 6, 8};

Инициализатор одномерного массива должен состоять из последовательности выражений, позволяющих выполнять присваивание для типа элементов массива. Эти выражения инициализируют элементы массива в порядке по возрастанию, начиная с элемента с нулевым индексом. Число выражений в инициализаторе массива определяет длину создаваемого экземпляра массива. Например, приведенный выше инициализатор массива создает экземпляр массива int[] с длиной 5, а затем инициализирует этот экземпляр следующими значениями:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

Инициализатор многомерного массива должен иметь столько же уровней вложенности, сколько измерений он содержит. Внешний уровень указывается самым левым измерением, внутренний самым правым, остальные измерения указывают промежуточные уровни в соответствующем порядке. Длина каждого из измерений массива определяется количеством элементов на соответствующем уровне вложенности инициализатора массива. Количество элементов каждого инициализатора вложенного массива должно быть равно количеству элементов в других инициализаторах того же уровня. Пример:

int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};

создает двумерный массив с длиной пять для левого измерения и с длиной два для правого измерения

int[,] b = new int[5, 2];

а затем инициализирует экземпляр массива следующими значениями:

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;

Если измерение, за исключением самого правого, имеет нулевую длину, предполагается, что все следующие измерения также имеют нулевую длину. Пример:

int[,] c = {};

создает двухмерный массив с нулевой длиной для самого правого и самого левого измерения:

int[,] c = new int[0, 0];

Если при создании массива выражение содержит как явное указание длины измерений, так и инициализатор массива, длина должна быть выражена в константах, а количество элементов на каждом уровне вложенности должно совпадать с длиной соответствующего измерения. Примеры:

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

В этом примере инициализатор для переменной y приведет к возникновению ошибки компилирования, поскольку выражение длины массива не является константой. Инициализатор для переменной z также приведет к возникновению ошибки компилирования, поскольку заданная длина и число элементов в инициализаторе не совпадают.

# Интерфейсы

В интерфейсе определяется контракт. Класс или структура, в которых реализуется этот интерфейс, должны соблюдать условия данного контракта. Интерфейс может наследовать от нескольких базовых интерфейсов, а в классе или структуре может быть реализовано несколько интерфейсов.

Интерфейсы могут содержать методы, свойства, события и индексаторы. Сам интерфейс не содержит реализации для определяемых им членов. Интерфейс всего лишь указывает члены, которые должны быть определены в классах или структурах, реализующих этот интерфейс.

## Объявления интерфейсов

Объявление\_интерфейса является объявлением\_типа (§9.6), где объявляется новый тип интерфейса.

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

Объявление\_интерфейса состоит из необязательного набора атрибутов (§17), за которым следуют необязательный набор модификаторов\_интерфейса (§13.1.1), необязательный модификатор partial, ключевое слово interface и идентификатор, именующий интерфейс, необязательная спецификация списка*\_параметров\_типа\_варианта* (§13.1.3), необязательная спецификация базы\_интерфейса (§13.1.4), необязательная спецификация предложений\_ограничений\_параметров\_типов (§10.1.5) и тело\_интерфейса (§13.1.5), которое может завершаться точкой с запятой.

### Модификаторы интерфейса

Объявление\_интерфейса может включать последовательность модификаторов интерфейса:

interface-modifiers:  
interface-modifier  
interface-modifiers interface-modifier

interface-modifier:  
new  
public  
protected  
internal  
private

Включение одного модификатора в объявление интерфейса несколько раз приведет к возникновению ошибки при компилировании.

Модификатор new разрешен только в том случае, если интерфейс определяется внутри класса. Он указывает, что в интерфейсе скрыт унаследованный член с таким же именем (см. §10.3.4).

Модификаторы public, protected, internal и private управляют доступом к интерфейсу. Допустимые модификаторы определяются контекстом объявления интерфейса (§3.5.1).

### Модификатор partial

Модификатор partial указывает, что данное объявление\_интерфейса является частичным объявлением типа. Несколько частичных объявлений интерфейса с одним именем в едином пространстве имен или объявлении типа объединяются в одно объявление интерфейса с соблюдением правил, приведенных в §10.2.

### Списки параметров типа варианта

Только типы интерфейсов и типы делегатов могут иметь списки параметров типа варианта. Отличие от обычных списков\_параметров\_типа заключается в необязательной аннотации\_варианта для каждого параметра типа.

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

Если аннотация варианта имеет значение out, тип параметра называется ковариантным. Если аннотация варианта имеет значение in, тип параметра называется контрвариантным. Если аннотации варианта нет, тип параметра называется инвариантным.

В этом примере

interface C<out X, in Y, Z>   
{  
 X M(Y y);

Z P { get; set; }  
}

X является ковариантным, Y – контрвариантным, а Z – инвариантным.

#### Безопасность вариативности

Появление аннотаций вариативности в списке параметров типа ограничивает количество местоположений, где могут появляться типы в рамках объявления типа.

Тип T является небезопасным при выводе, если справедливо одно из следующих условий.

* T является контравариантным типом параметра;
* T — это тип массива с типом элементов, небезопасным при выводе.
* T является типом интерфейса или делегата S<A1,… AK>, сформированным из универсального типа S<X1, .. XK>, в котором по меньшей мере один Ai содержит один из следующих вариантов:
* Xi является ковариантным или инвариантным, а Ai является небезопасным при выводе.
* Xi является ковариантным или инвариантным, а Ai является безопасным при выводе.

Тип T является небезопасным при вводе, если справедливо одно из следующих условий.

* T является параметром ковариантного типа.
* T является типом массива с типом элементов, небезопасным при вводе.
* T является типом интерфейса или делегата S<A1,… AK>, сформированным из универсального типа S<X1, .. XK>, в котором по меньшей мере один Ai содержит один из следующих вариантов:
* Xi является ковариантным или инвариантным, а Ai является небезопасным при вводе.
* Xi является контравариантным или инвариантным, а Ai является безопасным при выводе.

С интуитивной точки зрения, тип, небезопасный при выводе, запрещен в позиции вывода, а тип, небезопасный при вводе, запрещен в позиции ввода.

Тип является безопасным при выводе, если он не является небезопасным при выводе, а тип, безопасный при вводе, — это тип, который не является небезопасным при вводе.

#### Вариантные преобразования

Назначение аннотаций вариативности – предоставить более мягкие преобразования (но безопасные для типа) типам интерфейса и делегатов. Определения неявных (§6.1) и явных (§6.2) преобразований используют указание возможности преобразования вариативности, которое определяется следующим образом:

Тип T<A1, …, An> является вариативно-преобразуемым в тип T<B1, …, Bn>, если T является типом интерфейса или делегата, объявленного с параметрами типа варианта T<X1, …, Xn>, и для каждого параметра типа варианта Xi имеет смысл один из следующих пунктов.

* Xi является ковариантным, и существует неявное преобразование идентификации или ссылки из Ai в Bi.
* Xi является контравариантным, и существует неявное преобразование идентификации или ссылки из Bi в Ai.
* Xi является инвариантным и существует преобразование идентификации из Ai в Bi.

### Базовые интерфейсы

Интерфейс может не иметь предков или наследоваться из одного или нескольких интерфейсов, которые называются явными базовыми интерфейсами этого интерфейса. Если интерфейс имеет один или несколько явных базовых интерфейсов, в объявлении этого интерфейса за его идентификатором следует список базовых интерфейсов, разделяемых точкой с запятой.

interface-base:  
: interface-type-list

Для интерфейса сформированного типа явные базовые интерфейсы образуются путем замены объявлений явных базовых интерфейсов универсального типа для каждого параметра типа в объявлении базового интерфейса соответствующими аргументами типа сформированного типа.

Уровень доступа явных базовых интерфейсов для конкретного интерфейса должен быть не ниже уровня доступа самого интерфейса (§3.5.4). Например, указание интерфейса с уровнем доступа private или internal в качестве базы интерфейса для интерфейса с уровнем доступа public приведет к ошибке времени компиляции.

Явное или неявное наследование интерфейса из самого себя также приведет к возникновению ошибки компилирования.

Базовыми интерфейсами для интерфейса являются явные базовые интерфейсы и их базовые интерфейсы. Другими словами, набор базовых интерфейсов является полным транзитивным замыканием явных базовых интерфейсов, их явных базовых интерфейсов и так далее. Интерфейс наследует все члены своих базовых интерфейсов. В этом примере

interface IControl  
{  
 void Paint();  
}

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

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

interface IComboBox: ITextBox, IListBox {}

базовые интерфейсы интерфейса IComboBox — IControl, ITextBox и IListBox.

Другими словами, приведенный выше интерфейс IComboBox наследует члены SetText, SetItems и Paint.

Каждый базовый интерфейс любого интерфейса должен быть безопасным при выводе (§13.1.3.1). Класс или структура, которые реализуют интерфейс, также неявно реализуют все его базовые интерфейсы.

### Тело интерфейса

В теле интерфейса определяются члены этого интерфейса.

interface-body:  
{ interface-member-declarationsopt }

## Члены интерфейса

К членам интерфейса относятся члены, унаследованные из базовых интерфейсов, а также члены, объявленные в самом интерфейсе.

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

В объявлении интерфейса может содержаться любое количество членов, в том числе ни одного. Члены интерфейса должны быть методами, свойствами, событиями или индексами. Интерфейс не может содержать константы, поля, операторы, конструкторы экземпляров, деструкторы и типы, а также статические члены любого вида.

Члены интерфейса неявно имеют уровень доступа «public». Включение модификаторов в объявления членов интерфейса приведет к возникновению ошибки компилирования. В частности, члены интерфейсов не могут объявляться с модификаторами abstract, public, protected, internal, private, virtual, override и static.

Пример:

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; }  
}

объявляется интерфейс, который содержит по одному из всех допустимых видов членов: метод, свойство, событие и индекс.

Объявление интерфейса создает новую область объявления (§3.3), а объявления членов интерфейса, содержащиеся непосредственно в объявлении интерфейса, вводят в эту область объявления новых членов. К объявлениям членов интерфейса применяются следующие правила:

* Имя метода должно отличаться от имен всех свойств и событий, объявленных в том же интерфейсе. Помимо этого подпись (§3.6) метода должна отличаться от подписей всех других методов, объявленных в том же интерфейсе, а два метода, объявленные в одном интерфейсе, не могут иметь подписи, отличающиеся только модификаторами ref и out.
* Имя свойства или события должно отличаться от имен всех остальных членов, объявленных в том же интерфейсе.
* Подпись индексатора должна отличаться от подписей всех остальных индексаторов, объявленных в том же интерфейсе.

Унаследованные члены интерфейса не относятся к области объявления этого интерфейса. Таким образом, в интерфейсе разрешается объявить член с именем или подписью унаследованного члена. В этом случае говорят, что член производного интерфейса скрывает члена базового интерфейса. Скрытие унаследованного члена не считается ошибкой, но приводит к отображению компилятором предупреждения. Чтобы не выводить такие предупреждения, объявление члена производного интерфейса должно содержать модификатор new, который указывает, что данный производный член должен скрывать базовый член. Эта тема более подробно рассматривается в §3.7.1.2.

Если модификатор new содержится в объявлении, где не выполняется скрытие унаследованного члена, выводится соответствующее предупреждение. Чтобы не выводить такие предупреждения, следует удалить модификатор new.

Обратите внимание, что члены в классе object не являются, строго говоря, членами какого-либо интерфейса (§13.2). Тем не менее, члены в классе object доступны при поиске членов интерфейса любого типа (§7.4).

### Методы интерфейса

Методы интерфейса объявляются с использованием объявлений методов интерфейса:

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

В объявлении метода интерфейса атрибуты, тип возвращаемого значения, идентификатор и список формальных параметров имеют такой же смысл, что и в объявлении метода класса (§10.6). В объявлении метода интерфейса не разрешается указывать тело метода, в связи с этим объявление всегда завершается точкой с запятой.

Каждый формальный тип параметра метода интерфейса должен быть безопасным при вводе (§13.1.3.1), а тип возвращаемого значения – void или безопасный при выводе. Более того, каждое ограничение типа класса, типа интерфейса и типа параметра на любой тип параметра метода должно быть безопасным при вводе.

Эти правила обеспечивают безопасность типа при каждом ковариантном или контравариантном использовании интерфейса. Например,

interface I<out T> { void M<U>() where U : T; }

является недействительным, поскольку использование T в качестве ограничения параметра типа на U небезопасно при вводе.

Если бы это ограничение не располагалось в этом месте, стало бы возможным нарушение безопасности типа следующим образом:

class B {}  
class D : B {}  
class E : B {}  
class C : I<D> { public void M<U>() {…} }  
…  
I<B> b = new C();  
b.M<E>();

Это означает вызов C.M<E>. Но данный вызов требует, чтобы E было производным от D, таким образом безопасность типа будет под угрозой.

### Свойства интерфейса

Свойства интерфейса объявляются с использованием объявлений свойств интерфейса:

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

В объявлении свойства интерфейса атрибуты, тип и идентификатор имеют такой же смысл, что и в объявлении свойства класса (§10.7).

Методы доступа в объявлении свойства интерфейса соответствуют методам доступа в объявлении свойства класса (§10.7.2) за тем исключением, что тело метода доступа должно всегда быть точкой с запятой. Таким образом, методы доступа просто указывают, каким является свойство: доступным на чтение и запись, только на чтение или только на запись.

Тип свойства интерфейса должен быть безопасным при выводе, если существует метод доступа get, а также безопасным при вводе, если существует метод доступа set.

### События интерфейса

События интерфейса объявляются с использованием объявлений событий интерфейса:

interface-event-declaration:  
attributesopt newopt event type identifier ;

В объявлении события интерфейса атрибуты, тип и идентификатор имеют такой же смысл, что и в объявлении события класса (§10.8).

Тип события интерфейса должен быть безопасным при вводе.

### Индексаторы интерфейса

Индексаторы интерфейса объявляются с использованием объявлений индексаторов интерфейса:

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

В объявлении индексатора интерфейса атрибуты, тип и список формальных параметров имеют такой же смысл, что и в объявлении индексатора класса (§10.9).

Методы доступа в объявлении индексатора интерфейса соответствуют методам доступа в объявлении индексатора класса (§10.9) за тем исключением, что тело метода доступа должно всегда быть точкой с запятой. Таким образом, методы доступа просто указывают, каким является индекс: доступным на чтение и запись, только на чтение или только на запись.

Все типы формальных параметров индексатора интерфейса должны быть безопасны при вводе. Кроме того, любой из типов формальных параметров out или ref также должен быть безопасным при выводе. Обратите внимание, что даже параметры out должны быть безопасными при вводе, согласно ограничению базовой платформы выполнения.

Тип индексатора интерфейса должен быть безопасным при выводе, если существует метод доступа get, а также безопасным при вводе, если существует метод доступа set.

### Доступ к членам интерфейса

Доступ к членам интерфейса осуществляется с использованием выражений доступа к членам (§7.6.4) и доступа к индексаторам (§7.6.6.2) в форме I.M и I[A], где I указывает тип интерфейса, M — метод, свойство или событие этого типа интерфейса, а A — список аргументов индексатора.

Для интерфейсов, которые наследуют строго от одного предка (каждый интерфейс в цепочке наследования имеет только один прямой базовый интерфейс или не имеет его вовсе), результаты применения правил поиска членов (§7.4), вызова методов (§7.6.5.1) и доступа к индексатору (§7.6.6.2) аналогичны результатам для классов и структур: производные члены скрывают соответствующие базовые члены с тем же именем или той же подписью. Однако в интерфейсах с множественным наследованием могут происходить неоднозначности, связанные с объявлением в двух (или более) базовых интерфейсах методов с одинаковыми именами или подписями. В данном разделе приводится несколько примеров подобных ситуаций. Во всех случаях для разрешения таких неоднозначностей можно использовать явное приведение типов.

В этом примере

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  
 }  
}

выполнение первых двух операторов ведет к ошибкам при компилировании, поскольку поиск членов (§7.4) метода Count в интерфейсе IListCounter является неоднозначным. Как показано в этом примере, неоднозначность разрешается путем приведения переменной x к типу соответствующего базового интерфейса. Такое приведение не требует дополнительных затрат ресурсов во время выполнения — оно просто означает, что при компилировании экземпляр будет рассматриваться как находящийся на предыдущем уровне наследования.

В этом примере

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  
 }  
}

при вызове метода n.Add(1) выбирается метод IInteger.Add в результате применения правил разрешения перегрузки, рассматриваемых в разделе §7.5.3. Аналогичным образом, при вызове метода n.Add(1.0) выбирается метод IDouble.Add. Если вставлено явное приведение типов, кандидатом является только один метод, что позволяет избежать неоднозначности.

В этом примере

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  
 }  
}

член IBase.F скрыт членом ILeft.F. В вызове d.F(1) выбирается член ILeft.F, хотя имя IBase.F не скрыто в пути доступа, который ведет через интерфейс IRight.

Интуитивное правило скрытия членов в интерфейсах с множественным наследованием звучит так: если член скрыт в одном из путей доступа, он скрыт и во всех остальных путях доступа. Поскольку член IBase.F скрыт в пути доступа от IDerived к ILeft и к IBase, этот член также скрыт в пути доступа от IDerived к IRight и к IBase.

## Полные имена членов интерфейса

Иногда ссылка на член интерфейса имеет вид полного имени. Полное имя члена интерфейса состоит из имени интерфейса, в котором объявлен данный член, за которым следуют точка и имя нужного члена. Полное имя члена указывает на интерфейс, в котором объявлен этот член. Например, в объявлениях

interface IControl  
{  
 void Paint();  
}

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

полное имя каждого элемента Paint — IControl.Paint, а полное имя SetText — ITextBox.SetText.

В приведенном выше примере нельзя обращаться к Paint как ITextBox.Paint.

Если интерфейс является частью пространства имен, полное имя члена интерфейса включает имя этого пространства имен. Пример

namespace System  
{  
 public interface ICloneable  
 {  
 object Clone();  
 }  
}

Здесь полным именем метода Clone является System.ICloneable.Clone.

## Реализация интерфейсов

Интерфейсы реализуются в классах и структурах. Чтобы указать, что класс или структура непосредственно реализуют интерфейс, в список базовых классов такого класса или структуры следует включить идентификатор интерфейса. Пример:

interface ICloneable  
{  
 object Clone();  
}

interface IComparable  
{  
 int CompareTo(object other);  
}

class ListEntry: ICloneable, IComparable  
{  
 public object Clone() {...}

public int CompareTo(object other) {...}  
}

Класс или структура, которые непосредственно реализуют интерфейс, также непосредственно и неявно реализуют все его базовые интерфейсы. Это справедливо даже в том случае, если в списке базовых классов соответствующего класса или структуры не указаны явным образом все базовые интерфейсы. Пример:

interface IControl  
{  
 void Paint();  
}

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

class TextBox: ITextBox  
{  
 public void Paint() {...}

public void SetText(string text) {...}  
}

Здесь класс TextBox одновременно реализует интерфейсы IControl и ITextBox.

Если класс C непосредственно реализует интерфейс, все классы, производные от C, также неявно реализуют интерфейс. Базовые интерфейсы, указанные в объявлении класса, могут относиться к интерфейсам сформированного типа (§4.4). Базовый интерфейс не может быть параметром типа сам по себе, хотя он может включать параметры типа, находящиеся в соответствующей области. В следующем коде демонстрируется реализация и расширение сформированных типов классом:

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

Базовые интерфейсы объявления универсального класса должны удовлетворять правилу уникальности, которое изложено в §13.4.2.

### Явные реализации членов интерфейса

Для целей реализации интерфейсов в классе или структуре могут объявляться явные реализации членов интерфейса. Явная реализация члена интерфейса представляет собой объявление метода, свойства, события или индекса, содержащее ссылку на полное имя члена интерфейса. Пример

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) {...}  
}

В этом примере строки IDictionary<int,T>.this и IDictionary<int,T>.Add являются явными реализациями членов интерфейса.

В некоторых случаях имя члена интерфейса может быть неподходящим для реализующего класса. В этом случае член интерфейса может быть реализован с использованием явной реализации члена интерфейса. Например, в классе, который реализует абстракцию файла, скорее всего будет реализована функция-член Close, в результате выполнения этой функции высвобождается файловый ресурс и реализуется метод Dispose интерфейса IDisposable с использованием явной реализации члена интерфейса:

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);  
 }  
}

Получить доступ к явной реализации члена интерфейса с использованием его полного имени в вызванном методе, обращении к свойству или индексатору невозможно. Доступ к явной реализации члена интерфейса можно получить только через экземпляр интерфейса, в этом случае обращение к нему следует выполнять просто по имени его члена.

Включение в явную реализацию члена интерфейса модификаторов доступа, а также модификаторов abstract, virtual, override и static приведет к ошибке при компилировании.

Характеристики уровня доступа в явных реализациях члена интерфейса отличаются от таких характеристик других членов. Поскольку доступ к явным реализациям члена интерфейса по полному имени при вызове метода или при обращении к свойству невозможен, они в некотором роде являются закрытыми. Тем не менее, так как можно получить доступ к ним через экземпляр интерфейса, они также в некотором роде являются открытыми.

Явные реализации члена интерфейса служат двум основным целям:

* Поскольку к явным реализациям члена интерфейса нельзя получить доступ через экземпляры классов или структур, они позволяют исключить реализации интерфейса из открытого интерфейса класса или структуры. Это особенно полезно в том случае, если в классе или структуре реализуется внутренний интерфейс, который не предоставляет интереса для объекта, где используется этот класс или эта структура .
* Явные реализации члена интерфейса позволяют разрешить неоднозначности членов интерфейса с одинаковой подписью. Без явных реализаций члена интерфейса было бы невозможно иметь разные реализации члена интерфейса с одинаковыми подписью и типом возвращаемого значения, а также было бы невозможно включить в класс или структуру любые реализации членов интерфейса с одинаковой подписью и разными типами возвращаемых значений.

Явная реализация члена интерфейса является допустимой только в том случае, если в списке базовых классов класса или структуры содержится интерфейс, содержащий член, у которого полное имя, тип и типы параметров полностью совпадают с соответствующими данными явной реализации члена интерфейса. Так, в следующем классе

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}

int IComparable.CompareTo(object other) {...} // invalid  
}

объявление метода IComparable.CompareTo приведет к ошибке времени компиляции, поскольку интерфейс IComparable не включен в список базовых классов класса Shape и не является базовым интерфейсом для интерфейса ICloneable. Аналогичным образом в объявлениях

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}  
}

class Ellipse: Shape  
{  
 object ICloneable.Clone() {...} // invalid  
}

объявление метода ICloneable.Clone в классе Ellipse приведет к ошибке времени компиляции, поскольку интерфейс ICloneable не указан явно в списке базовых классов класса Ellipse.

Полное имя интерфейса должно указывать интерфейс, в котором был объявлен этот член. Так, в объявлениях

interface IControl  
{  
 void Paint();  
}

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

class TextBox: ITextBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}  
}

явная реализация члена интерфейса Paint должна быть записана в форме IControl.Paint.

### Уникальность реализованных интерфейсов

Интерфейсы, реализованные в объявлении уникального типа, должны оставаться уникальными для всех сформированных типов. Если бы это правило не действовало, было бы невозможно определить правильный метод для вызова в конкретных сформированных типах. Например, предположим, что объявление универсального класса может иметь следующий вид:

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() {...}  
}

В результате было бы невозможно определить, какой код должен выполняться в следующем случае:

I<int> x = new X<int,int>();  
x.F();

Чтобы определить, является ли список объявлений универсального типа допустимым, требуется выполнить следующие шаги:

* L — список интерфейсов, непосредственно определенных в универсальном объявлении класса, структуры или интерфейса C.
* Добавить в L все базовые интерфейсы для интерфейсов, уже содержащихся в L.
* Удалить из L все дубликаты.
* Если после создания на основе C любого возможного типа в результате подстановки в L аргументов типа два интерфейса в L станут идентичными, такое объявление C является недопустимым. При определении всех возможных сформированных типов не учитываются объявления ограничений.

В приведенном выше объявлении класса X список интерфейсов L состоит из интерфейсов I<U> и I<V>. Объявление является недопустимым, поскольку в любом сформированном типе, где U и V имеют одинаковый тип, эти два интерфейса также будут иметь одинаковый тип.

Можно объединять интерфейсы, находящиеся на разных уровнях наследования:

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() {…}  
}

Этот код является допустимым, несмотря на то, что в классе Derived<U,V> реализуются как интерфейс I<U>, так и интерфейс I<V>. Код

I<int> x = new Derived<int,int>();  
x.F();

вызовет метод класса Derived, поскольку класс Derived<int,int> фактически повторно реализует I<int> (§13.4.6).

### Реализация универсальных методов

Если в универсальном методе неявно реализуется метод интерфейса, то ограничения, заданные для каждого из параметров типа метода, должны быть одинаковы в обоих объявлениях (после замены любых параметров типа интерфейса соответствующими аргументами типа), где параметры типа метода указываются их положением слева направо.

Тем не менее, если в универсальном методе явно реализуется метод интерфейса, ограничения для реализующего метода не допускаются. Эти ограничения наследуются из метода интерфейса

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  
}

Метод C.F<T> неявно реализует метод I<object,C,string>.F<T>. В этом случае в методе C.F<T> не требуется (и не допускается) указывать ограничение T: object, так как object является неявным ограничением для всех параметров типа. Метод C.G<T> неявно реализует метод I<object,C,string>.G<T>, поскольку ограничения совпадают с ограничениями, установленными в интерфейсе после замены параметров типа интерфейса соответствующими аргументами типа. Ограничение для метода C.H<T> является ошибочным, так как запечатанные типы ( в данном случае string) не могут использоваться как ограничения. Отсутствие ограничения также было бы ошибкой, поскольку ограничения в неявных реализациях метода интерфейса должны совпадать. Таким образом, выполнить неявную реализацию метода I<object,C,string>.H<T> невозможно. Этот метод интерфейса может быть реализован только с использованием явной реализации члена интерфейса:

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);  
 }  
}

В этом примере явная реализация члена интерфейса вызывает открытый метод с применением более мягких ограничений. Обратите внимание, что присваивание из t в s является допустимым, поскольку T наследует ограничение T: string, хотя это ограничение и не указано в исходном коде.

### Сопоставление интерфейсов

В классе или структуре должны содержаться реализации всех членов интерфейсов, включенных в список базовых классов такого класса или структуры. Процедура выявления реализаций членов интерфейса в реализующих их классе или структуре называется сопоставлением интерфейсов.

При сопоставлении интерфейсов для класса или структуры C обнаруживает реализацию каждого из членов интерфейса, указанных в списке базовых классов C. Реализация конкретного члена интерфейса I.M, где I указывает интерфейс, в котором объявлен член M, определяется путем проверки каждого класса или структуры S, начиная с C и последовательно проходя по каждому базовому классу C до выявления совпадения:

* Если S содержит объявление явной реализации члена интерфейса, которая соответствует I и M, этот член является реализацией члена I.M.
* В обратном случае, если S содержит объявление открытого члена, не являющегося статическим, который соответствует M, то реализацией члена I.M является этот член. При выявлении совпадения для нескольких членов не указывается, какой из них является реализацией члена I.M. Эта ситуация может произойти только в том случае, если S относится к сформированному типу, в котором два члена, согласно объявлению в универсальном типе, имеют разные подписи, однако аргументы типа делают их подписи идентичными.

Если не удается выявить реализации для всех членов всех интерфейсов, указанных в списке базовых классов C, возникает ошибка при компилировании. Обратите внимание, что члены интерфейса содержат члены, наследуемые из базовых интерфейсов.

При сопоставлении интерфейсов член класса A считается соответствующим члену класса B в указанных ниже случаях.

* A и B являются методами, у A и B совпадают имена, типы и списки формальных параметров.
* A и B являются свойствами, у A и B совпадают имена и типы, A имеет те же методы доступа, что и B (A может иметь дополнительные методы доступа, если этот член не относится к явным реализациям члена интерфейса).
* A и B являются событиями, у A и B совпадают имена и типы.
* A и B являются индексами, у A и B совпадают типы и списки формальных параметров, A имеет те же методы доступа, что и B (A может иметь дополнительные методы доступа, если этот член не относится к явным реализациям члена интерфейса).

Алгоритм сопоставления интерфейсов предполагает следующее:

* При определении члена класса или структуры, реализующего член интерфейса, явные реализации члена интерфейса имеют приоритет по сравнению с другими членами того же класса или структуры.
* В сопоставлении интерфейсов не участвуют члены, которые являются статическими или не относятся к открытым.

В этом примере

interface ICloneable  
{  
 object Clone();  
}

class C: ICloneable  
{  
 object ICloneable.Clone() {...}

public object Clone() {...}  
}

реализацией метода Clone интерфейса ICloneable становится член ICloneable.Clone класса C, поскольку явные реализации членов интерфейса имеют приоритет по отношению к другим членам.

Если в классе или структуре реализуется два или более интерфейса, содержащих член с одинаковым именем, типом и типами параметров, можно сопоставить каждый из таких членов интерфейса единственному члену класса или структуры. Пример

interface IControl  
{  
 void Paint();  
}

interface IForm  
{  
 void Paint();  
}

class Page: IControl, IForm  
{  
 public void Paint() {...}  
}

В этом примере методам Paint интерфейсов IControl и IForm сопоставляется метод Paint класса Page. Разумеется, для каждого из этих двух методов можно создать отдельную явную реализацию члена интерфейса.

Если в классе или структуре реализуется интерфейс, который содержит скрытые члены, некоторые члены необходимо реализовать с использованием явных реализаций члена интерфейса. Пример

interface IBase  
{  
 int P { get; }  
}

interface IDerived: IBase  
{  
 new int P();  
}

Для реализации этого интерфейса потребуется по меньшей мере одна явная реализация члена интерфейса в одной из приведенных ниже форм:

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() {...}  
}

Если в классе реализуется несколько интерфейсов с одним базовым интерфейсом, может существовать только одна реализация этого базового интерфейса. В этом примере

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) {...}  
}

невозможно иметь отдельные реализации интерфейса IControl, включенного в список базовых классов, — интерфейса IControl, наследуемого интерфейсом ITextBox, и интерфейса IControl, наследуемого интерфейсом IListBox. Для этих интерфейсов отсутствует указание на их различие. Наоборот, в реализациях интерфейсов ITextBox и IListBox используется одна и та же реализация интерфейса IControl, а класс ComboBox просто считается реализующим три интерфейса: IControl, ITextBox и IListBox.

Члены базового класса участвуют в сопоставлении интерфейсов. В этом примере

interface Interface1  
{  
 void F();  
}

class Class1  
{  
 public void F() {}

public void G() {}  
}

class Class2: Class1, Interface1  
{  
 new public void G() {}  
}

метод F класса Class1 используется в реализации интерфейса Interface1 в классе Class2.

### Наследование реализаций интерфейсов

Класс наследует все реализации интерфейсов, содержащиеся в его базовых классах.

Без явной повторной реализации интерфейса, в производном классе нельзя изменить сопоставление интерфейсов, унаследованных им из базовых классов. Например, в объявлениях

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public void Paint() {...}  
}

class TextBox: Control  
{  
 new public void Paint() {...}  
}

метод Paint класса TextBox скрывает метод Paint в классе Control, однако не меняет сопоставление Control.Paint с IControl.Paint, и вызовы метода Paint в экземплярах класса и экземплярах интерфейса приведет к следующим результатам:

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();

Тем не менее, при сопоставлении метода интерфейса с виртуальным методом в классе можно переопределить этот виртуальный метод в производных классах и изменить реализацию интерфейса. Например, изменение записи приведенных выше объявлений на

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public virtual void Paint() {...}  
}

class TextBox: Control  
{  
 public override void Paint() {...}  
}

приведет к следующим результатам

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();

Так как явные реализации метода интерфейса не могут быть объявлены виртуальными, переопределить явную реализацию члена интерфейса невозможно. Однако в явной реализации члена интерфейса можно вызвать другой метод, который может быть объявлен виртуальным, что позволяет переопределить его в производных классах. Пример

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() { PaintControl(); }

protected virtual void PaintControl() {...}  
}

class TextBox: Control  
{  
 protected override void PaintControl() {...}  
}

В этом примере в классах, являющихся производными класса Control, можно указать реализацию метода IControl.Paint путем переопределения метода PaintControl.

### Повторная реализация интерфейса

В классе, который наследует реализацию интерфейса, разрешается повторно реализовать этот интерфейс путем включения его в список базовых классов.

Для повторной реализации интерфейса действуют такие же правила сопоставления интерфейсов, что и для первой реализации интерфейса. Таким образом, унаследованное сопоставление интерфейсов не влияет на сопоставление интерфейсов, установленное в повторной реализации этого интерфейса. Например, в объявлениях

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() {...}  
}

class MyControl: Control, IControl  
{  
 public void Paint() {}  
}

тот факт, что в классе Control метод IControl.Paint сопоставлен методу Control.IControl.Paint, не влияет на повторную реализацию в классе MyControl, где метод IControl.Paint сопоставлен методу MyControl.Paint.

В процедуре сопоставления повторно реализованных интерфейсов участвуют унаследованные объявления открытых членов и унаследованные явные объявления членов интерфейса. Пример

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() {}  
}

В этом примере реализация интерфейса IMethods в классе Derived сопоставляет методы этого интерфейса методам Derived.F, Base.IMethods.G, Derived.IMethods.H и Base.I.

При реализации интерфейса в классе также неявно реализуются все базовые интерфейсы этого интерфейса. Аналогичным образом повторная реализация интерфейса неявно является повторной реализацией всех базовых интерфейсов этого интерфейса. Пример

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() {...}  
}

Здесь в повторной реализации интерфейса IDerived также повторно реализуется интерфейс IBase, в результате чего метод IBase.F сопоставлен методу D.F.

### Абстрактные классы и интерфейсы

В абстрактных классах, как и в классах, не являющихся абстрактными, необходимо указать реализацию всех членов интерфейсов, включенных в список их базовых классов. Тем не менее, в абстрактном классе разрешается сопоставлять методы интерфейса абстрактным методам. Пример

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 public abstract void F();  
 public abstract void G();  
}

В этом примере реализация интерфейса IMethods сопоставляет методы F и G абстрактным методам, которые должны быть переопределены в производных от C классах, не являющихся абстрактными.

Обратите внимание, что явные реализации члена интерфейса не могут быть абстрактными, однако в таких реализациях можно вызывать абстрактные методы. Пример

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();  
}

В этом примере в неабстрактными классах, производных от класса C, необходимо переопределить методы FF и GG, что обеспечит фактическую реализацию интерфейса IMethods.

# Перечисления

Перечисляемый тип является особым типом значения (§4.1), который объявляет набор именованных констант.

Пример:

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

объявляется перечисляемый тип с именем Color, содержащий члены Red, Green и Blue.

## Объявления перечислений

Объявление перечисления объявляет новый перечисляемый тип. Объявление перечисления начинается с зарезервированного слова enum и содержит определение имени, доступности, базового типа и членов перечисления.

enum-declaration:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

Каждый перечисляемый тип имеет соответствующий целый тип, называемый базовым типом перечисляемого типа. Этот базовый тип должен иметь возможность представлять все значения перечислителя, определенные в перечислении. Объявление перечисления может явно объявлять базовый тип byte, sbyte, short, ushort, int, uint, long или ulong. Обратите внимание, что тип char не может использоваться в качестве базового типа. Объявление перечисления, которое не содержит явное объявления базового типа, имеет базовый тип int.

Пример:

enum Color: long  
{  
 Red,  
 Green,  
 Blue  
}

Здесь объявляется перечисление с базовым типом long. Разработчик может выбрать использование базового типа long, как это показано в примере, чтобы иметь возможность использования значений, которые находятся в диапазоне типа long, но не входят в диапазон типа int, или чтобы сохранить этот выбор на будущее.

## Модификаторы перечисления

Объявление перечисления может, при необходимости, включать последовательность модификаторов перечисления:

enum-modifiers:  
enum-modifier  
enum-modifiers enum-modifier

enum-modifier:  
new  
public  
protected  
internal  
private

Возникает ошибка времени компиляции, если один и тот же модификатор встречается несколько раз в объявлении перечисления.

Модификаторы объявления перечисления имеют тот же смысл, что и модификаторы объявления класса (§10.1.1). Однако обратите внимание, что в объявлении перечисления не разрешается использовать модификаторы abstract и sealed. Перечисления не могут быть абстрактными и не допускают производные типы.

## Члены перечисления

Тело объявления перечисляемого типа определяет нуль или более членов перечисления, которые являются именованными константами перечисляемого типа. Два члена не могут иметь одинаковое имя.

enum-member-declarations:  
enum-member-declaration  
enum-member-declarations , enum-member-declaration

enum-member-declaration:  
attributesopt identifier  
attributesopt identifier = constant-expression

Каждый член перечисления имеет связанное с ним постоянное значение. Тип этого значения является базовым типом для содержащего его перечисления. Постоянное значение для каждого члена перечисления должно находиться в диапазоне базового типа для перечисления. Пример:

enum Color: uint  
{  
 Red = -1,  
 Green = -2,  
 Blue = -3  
}

приводит к ошибке времени компиляции, так как значения констант -1, -2 и –3 не находятся в диапазоне базового целого типа uint.

Несколько членов перечисления могут совместно использовать одно и то же связанное значение. Пример:

enum Color   
{  
 Red,  
 Green,  
 Blue,

Max = Blue  
}

Здесь показано перечисление, в котором два его члена — Blue и Max — имеют одно и то же связанное значение.

Связанное значение члена перечисления присваивается явно или неявно. Если объявление члена перечисления имеет инициализатор константного выражения, значение этого константного выражения, неявно преобразованное в базовый тип перечисления, является связанным значением члена перечисления. Если в объявлении члена перечисления нет инициализатора, его связанное значение устанавливается неявно следующим образом:

* если член перечисления является первым членом перечисления, объявленным в перечисляемом типе, его связанное значение равно нулю;
* в противном случае связанное значение члена перечисления получается увеличением на единицу связанного значения предшествующего ему в тексте члена перечисления. Это увеличенное значение должно находиться в диапазоне значений, который может быть представлен базовым типом, иначе возникает ошибка времени компиляции.

Пример:

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";  
 }  
 }  
}

Здесь выводятся на печать имена членов перечисления и их связанные значения. Результат выполнения примера:

Red = 0  
Green = 10  
Blue = 11

по следующим причинам:

* члену перечисления Red автоматически присваивается значение нуль (так как у него нет инициализатора и он является первым членом перечисления);
* члену перечисления Green явно присваивается значение 10;
* а члену перечисления Blue автоматически присваивается значение на единицу большее, чем у члена, предшествующего ему в тексте.

Связанное значение члена перечисления не может, прямо или косвенно, использовать значение его собственного связанного члена перечисления. За исключением этого ограничения зацикливания, инициализаторы членов перечисления могут свободно ссылаться на другие инициализаторы членов перечисления, независимо от их положения в тексте. В инициализаторе члена перечисления значения других членов перечисления всегда обрабатываются как имеющие тип их базового типа, так что не требуются приведения к типу при ссылке на другие члены перечисления.

Пример:

enum Circular  
{  
 A = B,  
 B  
}

приводит к ошибке времени компиляции, так как объявления A и B являются циклическими. Член A зависит от B явно, а член B зависит от A неявно.

Члены перечисления именуются и имеют область видимости совершенно аналогично полям в классах. Областью видимости члена перечисления является тело содержащего его перечисляемого типа. В этой области видимости на члены перечисления можно ссылаться по их простому имени. Из всего остального кода имя члена перечисления должно уточняться именем его перечисляемого типа. У членов перечисления нет какой-либо объявленной доступности — член перечисления доступен, если доступен содержащий его перечисляемый тип.

## Тип System.Enum

Тип System.Enum — это абстрактный базовый класс всех перечисляемых типов (он является особым и отличным от базового типа перечисляемого типа), и члены, унаследованные от System.Enum, доступны в любом перечисляемом типе. Преобразование упаковки (§4.3.1) возможно из любого перечисляемого типа в тип System.Enum, а также возможно преобразование распаковки (§4.3.2) из типа System.Enum в любой перечисляемый тип.

Обратите внимание, что тип System.Enum сам по себе не является перечисляемым типом. Скорее, это тип\_класса, из которого производятся все перечисляемые\_типы. Тип System.Enum наследуется от типа System.ValueType (§4.1.1), который, в свою очередь, наследуется от типа object. Во время выполнения значение типа System.Enum может быть равно null или являться ссылкой на упакованное значение любого перечисляемого типа.

## Значения перечисления и операции

Каждый перечисляемый тип определяет отдельный тип; требуется явное преобразование перечисления (§6.2.2), чтобы выполнить преобразование между перечисляемым типом и целым типом, или между двумя перечисляемыми типами. Набор значений перечисляемого типа может включать его члены, но не ограничивается только ими. А именно, любое значение базового типа перечисления может быть приведено к перечисляемому типу, и оно является отдельным допустимым значением этого перечисляемого типа.

Члены перечисления имеют тип содержащего их перечисляемого типа (кроме случаев, когда они входят в инициализаторы других членов перечисления: см. §14.3). Значением члена перечисления, объявленного в перечисляемом типе E со связанным значением v, является (E)v.

Со значениями перечисляемых типов могут использоваться следующие операторы: ==, !=, <, >, <=, >= (§7.10.5), бинарный + (§7.8.4), бинарный‑ (§7.8.5), ^, &, | (§7.11.2), ~ (§7.7.4), ++ и -- (§7.6.9 и §7.7.5).

Каждый перечисляемый тип автоматически наследуется от класса System.Enum (который, в свою очередь, наследуется от классов System.ValueType и object). Таким образом, унаследованные методы и свойства этого класса могут использоваться в значениях перечисляемого типа.

# Делегаты

Делегаты делают возможными сценарии, которые в других языках, таких как C++, Pascal и Modula, реализуются с помощью указателей на функции. Однако в отличие от указателей на функции языка C++, делегаты являются полностью объектно-ориентированными, и в отличие от указателей на функции-члены языка C++, делегаты инкапсулируют и экземпляр объекта, и метод.

Объявление делегата определяет класс, производный от класса System.Delegate. Экземпляр делегата инкапсулирует список вызова, являющийся списком из одного или более методов, на каждый из которых ссылаются как на объект, допускающий вызов. Для методов экземпляров объект, допускающий вызов, состоит из экземпляра и метода для этого экземпляра. Для статических методов объект, допускающий вызов, состоит только из метода. Вызов экземпляра делегата с соответствующим набором аргументов приводит к вызову каждого из допускающих вызов объектов делегата с данным набором аргументов.

Интересным и полезным свойством экземпляра делегата является то, что он не знает и не заботится о классах методов, которые инкапсулирует; имеет значение только совместимость этих методов (§15.1) с типом делегата. Это делает делегаты вполне пригодными для «анонимного» вызова.

## Объявления делегатов

Объявление\_делегата — это объявление\_типа (§9.6), которое объявляет новый тип делегата.

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

Возникает ошибка времени компиляции, если один и тот же модификатор встречается несколько раз в объявлении делегата.

Модификатор new допускается только для делегатов, объявленных в другом типе; в этом случае модификатор указывает, что такой делегат скрывает унаследованный член с тем же именем, как описано в §10.3.4.

Модификаторы public, protected, internal и private управляют доступом к типу делегата. В зависимости от контекста, в котором встречается объявление делегата, некоторые из этих модификаторов запрещены (§3.5.1).

Именем типа делегата является идентификатор.

Необязательный список\_формальных\_параметров определяет параметры делегата, а тип\_возвращаемого\_значения указывает тип возвращаемого значения делегата.

Необязательный список\_параметров\_типа\_варианта (§13.1.3) задает параметры типа для самого делегата.

Возвращаемый тип типа делегата должен быть void или безопасным при выводе (§13.1.3.1).

Все типы формальных параметров типа делегата должны быть безопасными при вводе. Кроме того, любой из типов параметров out или ref также должен быть безопасным при выводе. Обратите внимание, что даже параметры out должны быть безопасными при вводе, согласно ограничению базовой платформы выполнения.

Типы делегатов в C# являются эквивалентом имени, а не структурным эквивалентом. А именно, два разных типа делегатов с одинаковыми списками параметров и типом возвращаемого значения считаются разными типами делегатов. Однако экземпляры двух отдельных, но структурно эквивалентных типов делегатов могут сопоставляться как равные (§7.9.8).

Пример:

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) {...}  
}

Методы A.M1 и B.M1 совместимы с типами делегатов D1 и D2, так как у них одинаковый тип возвращаемого значения и список параметров; однако эти типы делегатов являются двумя разными типами, так что они не являются взаимозаменяемыми. Методы B.M2, B.M3 и B.M4 несовместимы с типами делегатов D1 и D2, так как у них разные типы возвращаемого значения или списки параметров.

Подобно другим объявлениям универсальных типов, должны быть заданы аргументы типа для создания сконструированного типа делегата. Типы параметров и тип возвращаемого значения сконструированного типа делегата создаются заменой каждого параметра типа в объявлении делегата соответствующим аргументом типа сконструированного типа делегата. Результирующие тип возвращаемого значения и типы параметров используются при определении, какие методы совместимы со сконструированным типом делегата. Пример:

delegate bool Predicate<T>(T value);

class X  
{  
 static bool F(int i) {...}

static bool G(string s) {...}  
}

Метод X.F совместим с типом делегата Predicate<int>, а метод X.G совместим с типом делегата Predicate<string>.

Объявление\_делегата — это единственный способ объявить тип делегата. Тип делегата является типом класса, производного от System.Delegate. Типы делегатов являются неявно sealed, поэтому никакие производные типы от типа делегата не допускаются. Недопустимо также наследовать от класса System.Delegate тип класса, отличного от класса делегата. Обратите внимание, что System.Delegate сам не является типом делегата; это тип класса, от которого производятся все типы делегатов.

Язык C# предоставляет специальный синтаксис для создания экземпляра и вызова делегата. За исключением создания экземпляра, любые операции, которые могут применяться к классу или экземпляру класса, могут также применяться к классу делегата или экземпляру делегата, соответственно. В частности, можно получить доступ к членам типа System.Delegate с помощью обычного синтаксиса доступа к члену.

Набор методов, инкапсулированных экземпляром делегата, называется списком вызова. Если экземпляр делегата создается (§15.2) из единственного метода, он инкапсулирует этот метод, и его список вызова содержит только одну запись. Однако, когда объединяются два непустых экземпляра делегата, их списки вызова связываются — в очередности: левый операнд, а затем правый операнд — для формирования нового списка вызова, содержащего две или более записей.

Делегаты объединяются с помощью бинарных операторов + (§7.8.4) и += (§7.17.2). Делегат можно удалить из объединения делегатов с помощью бинарных операторов - (§7.8.5) и -= (§7.17.2). Делегаты можно проверять на равенство (§7.10.8).

В следующем примере показано создание нескольких экземпляров делегата и их соответствующие списки вызова:

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  
 }

}

При создании экземпляров cd1 и cd2 каждый из них инкапсулирует один метод. При создании экземпляра cd3 его список вызова содержит два метода, M1 и M2 в этом порядке. Список вызова экземпляра cd4 содержит методы M1, M2 и M1 в этом порядке. Список вызова экземпляра cd5 содержит методы M1, M2, M1, M1 и M2 в этом порядке. Дополнительные примеры объединения (а также удаления) делегатов см. в §15.4.

## Совместимость делегатов

Метод или делегат M совместим с типом делегата D, если верны все следующие условия:

* D и M имеют одинаковое число параметров, и каждый параметр в D имеет такие же модификаторы ref или out, как и соответствующий параметр в M;
* для каждого параметра значения (параметр без модификатора ref или out) существует преобразование идентификации (§6.1.1) или неявное преобразование ссылки (§6.1.6) из типа параметра в D в соответствующий тип параметра в M;
* для каждого параметра ref или out тип параметра в D такой же, как тип параметра в M;
* существует преобразование идентификации или неявное преобразование ссылки из типа возвращаемого значения M в тип возвращаемого значения D.

## Создание экземпляра делегата

Экземпляр делегата создается выражением\_создания\_делегата (§7.6.10.5) или преобразованием в тип делегата. Вновь созданный экземпляр делегата затем ссылается на одно из следующего:

* на статический метод, на который имеется ссылка в выражении\_создания\_делегата, или
* на целевой объект (который не может иметь значение null) и метод экземпляра, на которые имеются ссылки в выражении\_создания\_делегата, или
* на другой делегат.

Пример:

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  
 }  
}

После создания экземпляры делегата всегда ссылаются на один и тот же целевой объект и метод. Помните, что когда объединяются два делегата или один делегат удаляется из другого делегата, в результате создается новый делегат со своим собственным списком вызова; списки вызова объединяемых или удаляемых делегатов остаются неизменными.

## Вызов делегата

Язык C# предоставляет специальный синтаксис для вызова делегата. Когда вызывается непустой экземпляр делегата, чей список вызова содержит одну запись, он вызывает один метод с теми же аргументами, которые были ему заданы, и возвращает то же самое значение, что метод, на который имеется ссылка. (Подробные сведения о вызове делегата см. в §7.6.5.3). Если при вызове такого делегата происходит исключение, и это исключение не перехватывается в вызванном методе, поиск предложения перехвата исключения продолжается в методе, вызвавшем делегат, как если бы этот метод напрямую вызвал метод, на который ссылался делегат.

Вызов экземпляра делегата, чей список вызова содержит несколько записей, продолжается путем вызова каждого из методов в списке вызова, одновременно, по порядку. Каждому из вызванных таким образом методов передается тот же набор параметров, заданный экземпляру делегата. Если такой вызов делегата включает параметры ссылок (§10.6.1.2), каждый вызов метода происходит со ссылкой на ту же самую переменную; изменения этой переменной, сделанные одним методом в списке вызова, будут видимы для методов, находящихся далее в списке вызова. Если вызов делегата включает выходные параметры или возвращаемое значение, их окончательное значение будет получено от вызова последнего делегата в списке.

Если при обработке вызова такого делегата происходит исключение, и это исключение не перехватывается в вызванном методе, поиск предложения перехвата исключения продолжается в методе, вызвавшем делегат, а все методы, указанные далее в списке вызова, не вызываются.

Попытка вызова экземпляра делегата, имеющего значение null, приводит к исключению типа System.NullReferenceException.

В следующем примере показано, как создавать экземпляры, объединять, удалять и вызывать делегаты:

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;  
 cd3(50); // call M1 then M2

cd3 -= cd2;  
 cd3(60); // call M1

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  
 }  
}

Как показано в выражении cd3 += cd1;, делегат может присутствовать в списке вызова несколько раз. В этом случае он просто вызывается один раз при каждом появлении. В таком списке вызова, как этот, при удалении делегата фактически удаляется его последнее появление в списке вызова.

Непосредственно перед выполнением последнего оператора, cd3 -= cd1;, делегат cd3 ссылается на пустой список вызова. Попытка удаления делегата из пустого списка (или удаления несуществующего делегата из непустого списка) не является ошибкой.

В результате получается:

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

# Исключения

Исключения в языке C# обеспечивают структурированный, единообразный и типобезопасный способ обработки состояний ошибки, как на системном уровне, так и на уровне приложения. Механизм исключения в языке C# вполне сходен с механизмом в языке C++, с несколькими важными отличиями:

* В C# все исключения должны быть представлены экземпляром типа класса, производным от System.Exception. В C++ для представления исключения может использоваться любое значение любого типа;
* в C# блок finally (§8.10) может использоваться для записи кода завершения, который выполняется как при нормальном выполнении, так и при исключительных состояниях. Такой код труден для написания в C++ без дублирования кода;
* в C# исключения системного уровня, такие как переполнение, деление на нуль и разыменование null, имеют хорошо определенные классы исключений и находятся на одном уровне с состояниями ошибки уровня приложения.

## Причины исключений

Исключения могут вызываться двумя разными способами.

* Оператор throw (§8.9.5) вызывает исключение немедленно и безусловно. Управление никогда не передается оператору, следующему за оператором throw.
* Определенные исключительные состояния, возникающие при обработке операторов и выражения C#, служат причиной исключения в определенных обстоятельствах, когда операцию не удается выполнить нормально. Например, операция целочисленного деления (§7.8.2) создает исключение System.DivideByZeroException, если знаменатель равен нулю. Список различных исключений, которые могут возникнуть таким образом, см. в §16.4.

## Класс System.Exception

Класс System.Exception является базовым типом для всех исключений. Этот класс имеет несколько примечательных свойств, которые совместно используются всеми исключениями:

* Message — свойство только для чтения, типа string, содержащее удобочитаемое описание причины исключения;
* InnerException — доступное только для чтения свойство типа Exception. Если его значение не NULL, оно ссылается на исключение, вызвавшее текущее исключение — то есть, текущее исключение было вызвано при обработке блока catch, обрабатывавшего исключение InnerException. Иначе его значение, равное NULL, указывает, что это исключение не было вызвано другим исключением. Количество объектов исключения, сцепленных вместе таким образом, может быть произвольным.

Значения этих свойств можно указывать в вызовах конструктора экземпляров для System.Exception.

## Обработка исключений

Исключения обрабатываются оператором try (§8.10).

Когда происходит исключение, система ищет ближайшее предложение catch, которое может обработать исключение, как это определено типом времени выполнения исключения. Сначала в текущем методе выполняется поиск лексически объемлющего оператора try, и связанные предложения catch оператора try рассматриваются по порядку. Если это не удается, в методе, вызвавшем текущий метод, выполняется поиск лексически объемлющего оператора try, который содержит точку вызова текущего метода. Этот поиск продолжается, пока не будет найдена конструкция catch, которая может обработать текущее исключение, назвав имя класса исключения такого же класса или базового класса в выполняемом типе созданного исключения. Конструкция catch, которая не именует класс исключения, может обрабатывать любое исключение.

Когда соответствующая конструкция catch найдена, система подготавливает передачу управления первому оператору конструкции catch. Прежде чем начинается выполнение конструкции catch, система сначала последовательно выполняет все предложения finally связанные с операторами try с большим уровнем вложенности, чем тот оператор, который вызвал исключение.

Если соответствующая конструкция catch не найдена, выполняется одно из следующих двух действий:

* Если поиск соответствующей конструкции catch доходит до статического конструктора (§10.12) или инициализатора статического поля, создается исключение System.TypeInitializationException в точке, запустившей вызов статического конструктора. Внутреннее исключение System.TypeInitializationException содержит то исключение, которое возникло первоначально;
* если поиск соответствующих предложений catch достигает кода, который первоначально запустил поток, выполнение потока завершается. Влияние такого завершения определяется реализацией.

Исключения, происходящие при выполнении деструктора, заслуживают особого упоминания. Если исключение происходит при выполнения деструктора и не перехватывается, выполнение этого деструктора завершается и вызывается деструктор базового класса (если он имеется). Если нет базового класса (как в случае типа object) или нет деструктора базового класса, исключение удаляется.

## Общие классы исключений

Следующие исключения вызываются определенными операциями языка C#.

|  |  |
| --- | --- |
| System.ArithmeticException | Базовый класс для исключений, происходящих при арифметических операциях, таких как System.DivideByZeroException и System.OverflowException. |
| System.ArrayTypeMismatchException | Вызывается при ошибке сохранения в массиве, оттого что фактический тип сохраняемого элемента несовместим с фактическим типом массива. |
| System.DivideByZeroException | Вызывается при попытке деления целого значения на нуль. |
| System.IndexOutOfRangeException | Вызывается при попытке индексировать массив индексом меньше нуля или выходящим за границы массива. |
| System.InvalidCastException | Вызывается при ошибке во время выполнения явного преобразования из базового типа или интерфейса в производный тип. |
| System.NullReferenceException | Вызывается, если ссылка null используется таким образом, что требуется объект, на который указывает ссылка. |
| System.OutOfMemoryException | Вызывается при неудачной попытке выделения памяти (с помощью оператора new). |
| System.OverflowException | Вызывается при переполнении в арифметической операции в контексте checked. |
| System.StackOverflowException | Вызывается при исчерпании стека выполнения из-за слишком большого числа отложенных вызовов методов; обычно это указывает на очень глубокую или неограниченную рекурсию. |
| System.TypeInitializationException | Вызывается, когда статический конструктор создает исключение и нет предложений catch для его перехвата. |

# Атрибуты

Многое в языке C# дает возможность программисту указывать декларативные сведения о сущностях, определенных в программе. Например, доступность метода в классе указывается путем добавления к нему модификаторов метода public, protected, internal и private.

Язык C# позволяет программистам создавать новые виды декларативных сведений, называемых атрибутами. Программист может присоединить эти атрибуты к различным программным сущностям и извлекать сведения атрибута в среде выполнения. Например, структура может определить атрибут HelpAttribute, который можно поместить в определенные программные элементы (такие как классы и методы), чтобы обеспечить сопоставление между этими программными элементами и соответствующей документацией.

Атрибуты определяются посредством объявления классов атрибутов (§17.1), которые могут иметь позиционные и именованные параметры (§17.1.2). Атрибуты вложены в сущности в программе C# с помощью спецификаций атрибутов (§17.2) и могут извлекаться во время выполнения как экземпляры атрибутов (§17.3).

## Классы атрибутов

Класс, который прямо или косвенно производится от абстрактного класса System.Attribute, является классом атрибута. Объявление класса атрибута определяет новый вид атрибута, который можно поместить в объявлении. Классы атрибутов принято именовать с помощью суффикса Attribute. При использовании атрибута суффикс может оставаться или опускаться.

### Использование атрибутов

Атрибут AttributeUsage (§17.4.1) применяется для описания, как использовать класс атрибута.

Атрибут AttributeUsage имеет позиционный параметр (§17.1.2), позволяющий классу атрибута указать те виды объявлений, в которых можно использовать атрибут. Пример:

using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]  
public class SimpleAttribute: Attribute   
{  
 ...  
}

В этом примере определяется класс атрибута с именем SimpleAttribute, который можно помещать только в объявления\_классов и объявления\_интерфейсов. Пример:

[Simple] class Class1 {...}

[Simple] interface Interface1 {...}

Показано несколько возможностей использования атрибута Simple. Хотя этот атрибут определен с именем SimpleAttribute, при его использовании можно опускать суффикс Attribute, то есть пользоваться коротким именем Simple. Таким образом, приведенный выше пример семантически эквивалентен следующему:

[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}

Атрибут AttributeUsage имеет именованный параметр (§17.1.2), называемый AllowMultiple, который указывает, можно ли задать атрибут более одного раза для данной сущности. Если параметр AllowMultiple для класса атрибута имеет значение True, то этот класс атрибута является классом атрибута многократного использования и его можно указывать для сущности более одного раза. Если параметр AllowMultiple для класса атрибута имеет значение False, то этот класс атрибута является классом атрибута однократного использования и его можно указывать для сущности не более одного раза.

Пример:

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; }  
 }  
}

определен класс атрибута многократного использования с именем AuthorAttribute. Пример:

[Author("Brian Kernighan"), Author("Dennis Ritchie")]   
class Class1  
{  
 ...  
}

Показано объявление класса с двумя использованиями атрибута Author.

Атрибут AttributeUsage имеет другой именованный параметр, называемый Inherited, который указывает, наследуется ли указанный для базового класса атрибут также классами, производными от этого базового класса. Если параметр Inherited для класса атрибута имеет значение True, этот атрибут наследуется. Если параметр Inherited для класса атрибута имеет значение False, этот атрибут не наследуется. Если параметр не установлен, его значение по умолчанию True.

Класс атрибута X, не имеющий вложенного в него атрибута AttributeUsage, как в следующем примере:

using System;

class X: Attribute {...}

эквивалентен следующему:

using System;

[AttributeUsage(  
 AttributeTargets.All,  
 AllowMultiple = false,  
 Inherited = true)  
]  
class X: Attribute {...}

### Позиционные и именованные параметры

Классы атрибутов могут иметь позиционные параметры и именованные параметры. Каждый открытый конструктор экземпляров для класса атрибута определяет допустимую последовательность позиционных параметров для этого класса атрибута. Каждое не статическое, общее, доступное на чтение и запись поле и свойство для класса атрибута определяет именованный параметр для класса атрибута.

Пример:

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 {...}  
 }  
}

определяется класс атрибута с именем HelpAttribute, имеющий один позиционный параметр url и один именованный параметр Topic. Хотя свойство Url не статическое и открытое, оно не определяет именованный параметр, так как не является доступным для чтения и записи.

Этот класс атрибута можно использовать следующим образом:

[Help("http://www.mycompany.com/.../Class1.htm")]  
class Class1  
{  
 ...  
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]  
class Class2  
{  
 ...  
}

### Типы параметров атрибута

Типы позиционных и именованных параметров для класса атрибута ограничены типами параметров атрибута, к которым относятся:

* один из следующих типов: bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort;
* Тип object.
* Тип System.Type.
* перечислимый тип, если и он имеет доступность public, и типы, в которые он вложен (если вложен), также имеют доступность public (§17.2);
* одномерные массивы указанных выше типов.

Аргумент конструктора или общее поле, которые не имеют один из этих типов, не могут использоваться в качестве позиционных или именованных параметров в спецификации атрибута.

## Спецификация атрибута

Спецификация атрибута — это применение предварительно определенного атрибута к объявлению. Атрибут является частью дополнительных декларативных сведений, задаваемых для объявления. Атрибуты могут задаваться в глобальной области видимости (чтобы задать атрибуты для содержащей сборки или модуля) и для объявления\_типа (§9.6), объявления\_члена\_класса (§10.1.5), объявления\_члена\_интерфейса (§13.2), объявления\_члена\_структуры (§11.2), объявления\_члена\_перечисления (§14.3), объявления\_метода\_доступа (§10.7.2), объявления\_метода\_доступа\_события (§10.8.1) и списка\_формальных\_параметров (§10.6.1).

Атрибуты задаются в разделах атрибутов. Раздел атрибутов состоит из пары квадратных скобок, которые окружают список из одного или более атрибутов, разделенных запятыми. Порядок указания атрибутов в таком списке и порядок размещения разделов, вложенных в одну и ту же программную сущность, не имеет значения. Например, спецификации атрибутов [A][B], [B][A], [A, B] и [B, A] эквивалентны.

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

Атрибут состоит из имени\_атрибута и необязательного списка позиционных и именованных аргументов. Позиционные аргументы (если они имеются) предшествуют именованным аргументам. Позиционный аргумент состоит из выражения\_аргумента\_атрибута; именованный аргумент состоит из имени, за которым следует знак равенства, далее следует выражение\_аргумента\_атрибута, которые все вместе ограничены теми же правилами, которые применяются для простого присваивания. Порядок именованных аргументов не имеет значения.

Имя\_атрибута определяет класс атрибута. Если имя\_атрибута имеет форму имени\_типа, это имя должно ссылаться на класс атрибута. Иначе возникает ошибка времени компиляции. Пример:

class Class1 {}

[Class1] class Class2 {} // Error

В результате возникает ошибка времени компиляции, так как предпринята попытка использовать Class1 в качестве класса атрибута, хотя Class1 не является классом атрибута.

Определенные контексты разрешают спецификацию атрибута более чем для одного целевого объекта. Программа может явно задать целевой объект включением описателя\_целевого\_объекта\_атрибута. Если атрибут размещен на глобальном уровне, требуется глобальный\_описатель\_целевого\_объекта\_атрибута. Во всех остальных расположениях применяется обоснованное значение по умолчанию, но описатель\_\_целевого\_объекта\_атрибута может использоваться для подтверждения или переопределения значения по умолчанию в определенных неоднозначных ситуациях (или просто для подтверждения значения по умолчанию в не вызывающих сомнения случаях). Таким образом, описатели\_целевого\_объекта\_атрибута обычно могут быть опущены, кроме случаев их использования на глобальном уровне. В случае потенциально неоднозначных контекстов решение принимается следующим образом:

* атрибут, заданный в глобальной области видимости, может применяться либо к конечной сборке, либо к конечному модулю. Для этого контекста нет значения по умолчанию, поэтому в этом контексте всегда требуется описатель\_целевого\_объекта\_атрибута. Наличие описателя\_\_целевого\_объекта\_атрибута assembly указывает, что атрибут применяется к конечной сборке; наличие описателя\_целевого\_объекта\_атрибута module указывает, что атрибут применяется к конечному модулю;
* атрибут, заданный в объявлении делегата, может применяться либо к объявляемому делегату, либо к его возвращаемому значению. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к делегату. Наличие описателя\_целевого\_объекта\_атрибута type указывает, что атрибут применяется к делегату; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению;
* атрибут, заданный в объявлении метода, может применяться либо к объявляемому методу, либо к его возвращаемому значению. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к методу. Наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к методу; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению;
* атрибут, заданный в объявлении оператора, может применяться либо к объявляемому оператору, либо к его возвращаемому значению. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к оператору. Наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к оператору; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению.
* атрибут, заданный в объявлении события, в котором опущены методы доступа к событиям, может применяться к объявляемому событию, к связанному полю (если событие не является абстрактным) или к связанным методам add и remove. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к событию. Наличие описателя\_целевого\_объекта\_атрибута event указывает, что атрибут применяется к событию; наличие описателя\_целевого\_объекта\_атрибута field указывает, что атрибут применяется к полю; а наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к методам.
* атрибут, заданный в объявлении метода доступа get для объявления свойства или индексатора, может применяться либо к связанному методу, либо к его возвращаемому значению. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к методу. Наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к методу; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению;
* атрибут, указанный в объявлении метода доступа set для объявления свойства или индексатора, может применяться либо к связанному методу, либо к его одиночному неявному параметру. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к методу. Наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к методу; наличие описателя\_целевого\_объекта\_атрибута param указывает, что атрибут применяется к параметру; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению.
* атрибут, заданный в объявлении метода доступа add или remove для объявления события, может применяться или к связанному методу, или к его одиночному параметру. При отсутствии описателя\_целевого\_объекта\_атрибута атрибут применяется к методу. Наличие описателя\_целевого\_объекта\_атрибута method указывает, что атрибут применяется к методу; наличие описателя\_целевого\_объекта\_атрибута param указывает, что атрибут применяется к параметру; наличие описателя\_целевого\_объекта\_атрибута return указывает, что атрибут применяется к возвращаемому значению.

В других контекстах включение описателя\_целевого\_объекта\_атрибута разрешено, но излишне. Например, объявление класса может или включать, или опускать описатель type:

[type: Author("Brian Kernighan")]  
class Class1 {}

[Author("Dennis Ritchie")]  
class Class2 {}

Является ошибкой задание недопустимого описателя\_целевого\_объекта\_атрибута. Например, описатель param нельзя использовать в объявлении класса:

[param: Author("Brian Kernighan")] // Error  
class Class1 {}

Классы атрибутов принято именовать с помощью суффикса Attribute. Имя\_атрибута в форме имени\_типа может или включать, или опускать этот суффикс. Если обнаружен класс атрибута и с этим суффиксом, и без суффикса, эта неоднозначность приводит к ошибке времени компиляции. Если имя\_атрибута записывается таким образом, что его самый правый идентификатор является буквальным идентификатором (§2.4.2), то сопоставляется только атрибут без суффикса, позволяя таким образом разрешить неоднозначность. Пример:

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 {}

показаны два класса атрибутов с именами X и XAttribute. Атрибут [X] является неоднозначным, так как он может ссылаться на X или на XAttribute. Использование буквального идентификатора позволяет указать точное намерение в таких редких случаях. Атрибут [XAttribute] не является неоднозначным (хотя он мог бы им быть, если бы имелся класс атрибута с именем XAttributeAttribute). Если удаляется объявление для класса X, то оба атрибута ссылаются на класс атрибута с именем XAttribute, как это показано в следующем примере:

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 {}

Использование класса атрибута одноразового использования более одного раза для одной и той же сущности является ошибкой времени компиляции. Пример:

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 {}

В результате возникает ошибка времени компиляции, так как предпринимается попытка использования HelpString, который является классом атрибута одноразового использования, более одного раза в объявлении Class1.

Выражение E является выражением\_аргумента\_атрибута, если верны все следующие утверждения:

* тип E — это тип параметра атрибута (§17.1.3);
* во время компиляции значение E может быть разрешено в один из следующих типов:
* постоянное значение;
* Объект System.Type.
* Одномерный массив выражения\_аргумента\_атрибута.

Пример:

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 {}

Выражение\_typeof (§7.6.11), используемое в качестве выражения аргумента атрибута, может ссылаться на неуниверсальный тип, закрытый сконструированный тип или на несвязанный универсальный тип, но не может ссылаться на открытый тип. Это обеспечивает возможность разрешения выражения во время компиляции.

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  
}

## Экземпляры атрибутов

Экземпляр атрибута — это экземпляр, представляющий атрибут во время выполнения. Атрибут определяется с помощью класса атрибута, позиционных аргументов и именованных аргументов. Экземпляр атрибута — это экземпляр класса атрибута, который инициализируется с помощью позиционных и именованных аргументов.

Извлечение экземпляра атрибута включает обработку как во время компиляции, так и во время выполнения, как описано в следующих разделах.

### Компиляция атрибута

Компиляция атрибута с классом атрибута T, списком\_позиционных\_аргументов P и списком\_именованных\_аргументов N состоит из следующих шагов:

* выполнение шагов обработки времени компиляции для компиляции выражения\_создания\_объекта вида new T(P). Результатом этих шагов является или ошибка времени компиляции, или определение конструктора экземпляров C для T, который может быть вызван во время выполнения;
* если у C нет доступности public, возникает ошибка времени компиляции;
* для каждого именованного аргумента Arg в N:
* разрешение Name быть идентификатором именованного аргумента Arg;
* Name должен идентифицировать нестатическое открытое с доступом для чтения и записи поле или свойство в T. Если в T нет такого поля или свойства, возникает ошибка времени компиляции.
* Сохранение следующих сведений для создания экземпляра атрибута во время выполнения: класс атрибута T, конструктор экземпляров C в T, список\_позиционных\_аргументов P и список\_именованных\_аргументов N.

### Извлечение экземпляра атрибута во время выполнения

Компиляция атрибута выдает класс атрибута T, конструктор экземпляров C в T, список\_позиционных\_аргументов P и список\_именованных\_аргументов N. Если даны эти сведения, экземпляр атрибута может быть извлечен во время выполнения с помощью следующих шагов:

* выполнение шагов обработки времени выполнения для выполнения выражения\_создания\_объекта вида new T(P) с помощью конструктора экземпляров C, как определено во время компиляции. Эти шаги приводят либо к исключению, либо к созданию экземпляра O объекта T;
* для каждого именованного аргумента Arg в N в следующем порядке:
* разрешение Name быть идентификатором именованного аргумента Arg; Если Name не идентифицирует нестатическое, открытое, с доступом для чтения и записи поле или свойство в O, вызывается исключение;
* разрешение Value быть результатом вычисления выражения\_аргумента\_атрибута Arg;
* если Name определяет поле в O, установить в этом поле значение Value;
* в противном случае Name определяет свойство в O. Задать для данного свойства значение Value;
* результатом является O, экземпляр класса атрибута T, инициализированный с помощью списка\_позиционных\_аргументов P и списка\_именованных\_аргументов N.

## Зарезервированные атрибуты

Небольшое число атрибутов некоторым образом влияют на язык. К этим атрибутам относятся:

* System.AttributeUsageAttribute (§17.4.1), используемый для описания способов, которыми можно использовать класс атрибута;
* System.Diagnostics.ConditionalAttribute (§17.4.2), используемый для определения условных методов;
* System.ObsoleteAttribute (§17.4.3), используемый для пометки члена как устаревшего;
* System.Runtime.CompilerServices.CallerLineNumberAttribute, System.Runtime.CompilerServices.CallerFilePathAttribute и System.Runtime.CompilerServices.CallerMemberNameAttribute (§17.4.4), используемые для передачи данных о вызывающем контексте необязательным параметрам.

### Атрибут AttributeUsage

Атрибут AttributeUsage используется для описания способа использования класса атрибута.

Класс, к которому добавлен атрибут AttributeUsage, должен прямо или косвенно наследоваться от класса System.Attribute. Иначе возникает ошибка времени компиляции.

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  
 }  
}

### Атрибут Conditional

Атрибут Conditional позволяет определять условные методы и классы условных атрибутов.

namespace System.Diagnostics  
{  
 [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,  
 AllowMultiple = true)]  
 public class ConditionalAttribute: Attribute  
 {  
 public ConditionalAttribute(string conditionString) {...}

public string ConditionString { get {...} }  
 }  
}

#### Условные методы

Метод, к которому добавлен атрибут Conditional, является условным методом. Атрибут Conditional указывает условие путем проверки символа условной компиляции. Вызовы условного метода или включаются, или опускаются, в зависимости от того, определен ли этот символ в точке вызова. Если символ определен, вызов включается; иначе вызов (включая оценку выражения-получателя и параметров вызова) опускается.

Для условного метода имеются следующие ограничения:

* Условный метод должен быть методом в объявлении\_класса или объявлении\_структуры. Если атрибут Conditional указан для метода в объявлении интерфейса, возникает ошибка времени компиляции.
* Условный метод должен иметь тип возвращаемого значения void.
* Условный метод не должен быть помечен модификатором override. Однако условный метод может быть помечен модификатором virtual. Переопределения такого метода являются неявно условными и они не должны явно помечаться атрибутом Conditional.
* условный метод не должен быть реализацией метода интерфейса. Иначе возникает ошибка времени компиляции.

Кроме того, ошибка времени компиляции возникает, если условный метод используется в выражении\_создания\_делегата. Пример:

#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();  
 }  
}

Class1.M объявляется как условный метод. Этот метод вызывается методом Test класса Class2. Так как символ условной компиляции DEBUG определен, если вызывается метод Class2.Test, он вызовет M. Если бы символ DEBUG не был определен, метод Class2.Test не вызвал бы Class1.M.

Обратите внимание, что включение или исключение вызова условного метода управляется символами условной компиляции в точке вызова. В этом примере

Файл class1.cs:

using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void F() {  
 Console.WriteLine("Executed Class1.F");  
 }  
}

Файл class2.cs:

#define DEBUG

class Class2  
{  
 public static void G() {  
 Class1.F(); // F is called  
 }  
}

Файл class3.cs:

#undef DEBUG

class Class3  
{  
 public static void H() {  
 Class1.F(); // F is not called  
 }  
}

каждый из классов Class2 и Class3 содержит вызовы условного метода Class1.F, вызов которого обусловленного тем, определен или нет символ DEBUG. Так как этот символ определен в контексте Class2, но не Class3, вызов F в Class2 включен, тогда как вызов F в Class3 опущен.

Использование условных методов в цепочке наследования может привести к путанице. Вызовы условного метода посредством base в виде base.M подчиняются обычным правилам вызова условного метода. В этом примере

Файл class1.cs:

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public virtual void M() {  
 Console.WriteLine("Class1.M executed");  
 }  
}

Файл class2.cs:

using System;

class Class2: Class1  
{  
 public override void M() {  
 Console.WriteLine("Class2.M executed");  
 base.M(); // base.M is not called!  
 }  
}

Файл class3.cs:

#define DEBUG

using System;

class Class3  
{  
 public static void Test() {  
 Class2 c = new Class2();  
 c.M(); // M is called  
 }  
}

Class2 включает вызов метода M, определенного в его базовом классе. Этот вызов опускается, так как базовый метод является условным, базированным на наличии символа DEBUG, который здесь не определен. Таким образом, метод записывает на консоль только сообщение "Class2.M executed". Разумное использование ПРО\_описаний может устранить такие проблемы.

#### Классы условных атрибутов

Класс атрибута (§17.1), к которому добавлен один или несколько атрибутов Conditional, является классом условных атрибутов. Таким образом, класс условных атрибутов связывается с символами условной компиляции, объявленными в его атрибутах Conditional. Рассмотрим следующий пример:

using System;  
using System.Diagnostics;  
[Conditional("ALPHA")]  
[Conditional("BETA")]  
public class TestAttribute : Attribute {}

объявлен атрибут TestAttribute в качестве класса условных атрибутов, связанного с символами условной компиляции ALPHA и BETA.

Спецификации атрибута (§17.2) условного атрибута включаются, если в точке спецификации определен один или несколько его связанных символов условной компиляции, иначе спецификация атрибута опускается.

Обратите внимание, что включение или исключение спецификации атрибута класса условных атрибутов управляется символами условной компиляции в точке спецификации. В этом примере

Файл test.cs:

using System;  
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}

Файл class1.cs:

#define DEBUG

[Test] // TestAttribute is specified

class Class1 {}

Файл class2.cs:

#undef DEBUG

[Test] // TestAttribute is not specified

class Class2 {}

к каждому из классов Class1 и Class2 добавляется атрибут Test, который является условным и зависит от того, определен или нет символ DEBUG. Так как этот символ определен в контексте Class1, но не в контексте Class2, спецификация атрибута Test для Class1 включена, тогда как спецификация атрибута Test для Class2 опущена.

### Атрибут Obsolete

Атрибут Obsolete используется для пометки типов и членов типов, которые не следует больше использовать.

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 {...} }  
 }  
}

Если программа использует тип или член, к которому добавлен атрибут Obsolete, компилятор выдает предупреждение или ошибку. А именно, компилятор выдает предупреждение, если не предоставлен параметр ошибки или если параметр ошибки имеет значение false. Компилятор выдает сообщение об ошибке, если указан параметр ошибки со значением true.

В этом примере

[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();  
 }  
}

к классу A добавляется атрибут Obsolete. Каждое использование A в Main приводит к выдаче предупреждения, в состав которого включено сообщение "Этот класс устаревший; используйте вместо него класс B".

### Атрибуты сведений о вызывающем коде

В целях ведения журналов и создания отчетов иногда бывает полезно, чтобы функция-член получала определенные сведения о вызывающем коде во время компиляции. Атрибуты сведений о вызывающем коде обеспечивает способ прозрачной передачи такой информации.

Если необязательный параметр аннотирован одним из атрибутов сведений о вызывающем коде, пропуск соответствующего аргумента в вызове не обязательно приводит к замене значения параметра по умолчанию. Напротив, если указанные сведения о вызывающем контексте доступны, они передаются в качестве значения аргумента.

Пример:

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);  
}

Вызов метода Log() без аргументов приведет к печати номера строки пути к файлу для вызова, а также имени члена, в котором этот вызов выполняется.

Атрибуты сведений о вызывающем коде могут содержаться в необязательных параметрах в любом месте программы, в том числе в объявлениях делегатов. Однако конкретные атрибуты сведений о вызывающем коде имеют ограничения, касающиеся типов параметров, к которым они могут применяться. Поэтому всегда выполняется неявное преобразование из типа подставляемого значения в тип параметра.

Применение одного атрибута сведений о вызывающем коде к параметру, содержащемуся одновременно в определяющей и реализующей частях объявления разделяемого метода, приводит к ошибке. Применяются только атрибуты сведений о вызывающем коде в определяющем объявлении, тогда как атрибуты сведений о вызывающем коде из реализующего объявления пропускаются.

Сведения о вызывающем коде не влияют на разрешение перегрузок. Если необязательные параметры с атрибутами отсутствуют в исходном вызывающем коде, эти параметры пропускаются при разрешении перегрузок точно так же, как пропускаются все другие отсутствующие необязательные параметры (§7.5.3).

Сведения о вызывающем коде подставляются только при явном вызове функции в исходном коде. Неявные вызовы, такие как неявные вызовы родительского конструктора, не имеют исходного местоположения и не приводят к подстановке сведений о вызывающем коде. Динамически привязанные вызовы также не подставляют сведения о вызывающем коде. Если в таких вызовах не указывается параметр с атрибутом сведений о вызывающем коде, вместо него используется заданное значение параметра по умолчанию.

Единственным исключением являются выражения запросов. Они считаются синтаксическими расширениями, и, если в расширяемых ими вызовах отсутствуют параметры с атрибутами сведений о вызывающем коде, сведения о вызывающем коде подставляются. В качестве местоположения используется местоположение предложения запроса, из которого был создан вызов.

Если для параметра указано несколько атрибутов сведений о вызывающем коде, они используются в следующем порядке: CallerLineNumber, CallerFilePath, CallerMemberName.

#### Атрибут CallerLineNumber

Атрибут System.Runtime.CompilerServices.CallerLineNumberAttribute разрешается применять к необязательным параметрам при наличии стандартного неявного преобразования (§6.3.1) из константного значения int.MaxValue в тип параметра. Это обеспечивает отсутствие ошибок при передаче любых неотрицательных номеров строк в это значение.

Если при вызове функции из некоторого местоположения в исходном коде не указывается необязательный параметр с атрибутом CallerLineNumberAttribute, то вместо значения параметра по умолчанию в качестве аргумента вызова используется числовой литерал, представляющий номер строки данного местоположения.

Если вызов распространяется на несколько строк, выбор строки зависит от конкретной реализации.

Обратите внимание, что на значение номера строки могут влиять директивы #line (§2.5.7).

#### Атрибут CallerFilePath

Атрибут System.Runtime.CompilerServices.CallerFilePathAttribute разрешается применять к необязательным параметрам при наличии стандартного неявного преобразования (§6.3.1) из типа string в тип параметра.

Если при вызове функции из некоторого местоположения в исходном коде не указывается необязательный параметр с атрибутом CallerFilePathAttribute, то вместо значения параметра по умолчанию в качестве аргумента вызова используется строковый литерал, представляющий путь к файлу для данного местоположения.

Формат пути к файлу зависит от реализации.

Обратите внимание, что на значение пути к файлу могут влиять директивы #line (§2.5.7).

#### Атрибут CallerMemberName

Атрибут System.Runtime.CompilerServices.CallerMemberNameAttribute разрешается применять к необязательным параметрам при наличии стандартного неявного преобразования (§6.3.1) из типа string в тип параметра.

Если при вызове функции из некоторого местоположения внутри тела функции-члена или внутри атрибута, применяемого к самой функции-члену или ее типу возвращаемого значения, в составе параметров или параметров типов исходного кода отсутствует необязательный параметр с атрибутом CallerMemberNameAttribute, то вместо значения параметра по умолчанию в качестве аргумента вызова используется строковый литерал, представляющий имя этого члена.

Для вызовов, осуществляемых из универсальных методов, используется только имя самого метода без списка параметров типов.

Для вызовов из явных реализаций членов интерфейса используется только имя самого метода без предшествующего имени интерфейса.

Для вызовов из методов доступа к свойствам или событиям в качестве имени члена используется имя самого свойства или события.

Для вызовов из методов доступа к индексатору в качестве имени члена используется имя, предоставляемое атрибутом IndexerNameAttribute (§17.5.2.1), или имя члена индексатора (при его наличии); в противном случае используется имя Item по умолчанию.

Для вызовов, осуществляемых из объявлений конструкторов экземпляров, использование имени члена статических конструкторов, деструкторов или операторов зависит от конкретной реализации.

## Атрибуты для взаимодействия

Примечание. Этот раздел применим только для реализации Microsoft .NET языка C#.

### Взаимодействие с компонентами COM и Win32

Исполняющая среда .NET предоставляет большое число атрибутов, позволяющих программам C# взаимодействовать с компонентами, написанными с использованием модели COM и библиотек DLL Win32. Например, атрибут DllImport можно использовать для метода static extern, чтобы указать, что реализацию метода следует искать в библиотеке DLL Win32. Эти атрибуты находятся в пространстве имен System.Runtime.InteropServices; подробную документацию по этим атрибутам можно найти в документации по среде выполнения .NET.

### Взаимодействие с другими языками .NET

#### Атрибут IndexerName

Индексаторы реализованы в .NET с помощью индексированных свойств и имеют имя в метаданных .NET. Если атрибут IndexerName отсутствует для индексатора, по умолчанию используется имя Item. Атрибут IndexerName позволяет разработчику переопределить это значение по умолчанию и задать другое имя.

namespace System.Runtime.CompilerServices.CSharp  
{  
 [AttributeUsage(AttributeTargets.Property)]  
 public class IndexerNameAttribute: Attribute  
 {  
 public IndexerNameAttribute(string indexerName) {...}

public string Value { get {...} }   
 }  
}

# Небезопасный код

Базовый язык C#, как определено в предыдущих главах, заметно отличается от C и C++ тем, что в нем не используются указатели в качестве типа данных. Вместо этого C# предоставляет ссылки и возможность создавать объекты, управляемые сборщиком мусора. Эта разработка в сочетании с другими особенностями делает язык C# гораздо более безопасным, чем C и C++. В базовом языке C# просто невозможно иметь неинициализированную переменную, «висящий» указатель или выражение, индексирующее массив за пределами его границ. Таким образом устраняются целые категории ошибок, регулярно досаждающих в программах на C и C++.

Хотя практически у каждой конструкции с типом указателя в C или C++ имеется аналог ссылочного типа в C#, тем не менее в некоторых ситуациях доступ к типам указателей становится необходимостью. Например, согласование с операционной системой, доступ к устройству, сопоставленному памяти, или реализация критического по времени алгоритма могут оказаться неосуществимыми без доступа к указателям. Для удовлетворения этой потребности C# предоставляет возможность писать небезопасный код.

В небезопасном коде можно объявлять и производить операции с указателями, выполнять преобразования между указателями и целыми типами, получать адреса переменных и так далее. В некотором смысле написание небезопасного кода очень похоже на написание кода C внутри программы на C#.

Небезопасный код в действительности является «безопасной» возможностью с точки зрения как разработчиков, так и пользователей. Небезопасный код должен быть явно помечен модификатором unsafe, так что разработчик не может случайно использовать небезопасные возможности, а механизм выполнения обеспечивает невозможность выполнения небезопасного кода в ненадежной среде.

## Небезопасные контексты

Небезопасные возможности C# доступны только в небезопасных контекстах. Небезопасный контекст вводится включением модификатора unsafe в объявление типа или члена, или использованием небезопасного\_оператора:

* объявление класса, структуры, интерфейса или делегата может включать модификатор unsafe, в этом случае все текстовое пространство этого объявления (включая тело класса, структуры или интерфейса) считается небезопасным контекстом;
* объявление поля, метода, свойства, события, индексатора, оператора, конструктора экземпляров, деструктора или статического конструктора может включать модификатор unsafe, в этом случае все текстовое пространство этого объявления члена считается небезопасным контекстом;
* небезопасный\_оператор дает возможность использовать небезопасный контекст внутри блока. Все текстовое пространство соответствующего блока считается небезопасным контекстом.

Соответствующие грамматические расширения показаны ниже. Для краткости все порождения, описанные в предыдущих главах, заменяются многоточием (...).

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:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement

unsafe-statement:  
unsafe block

В этом примере

public unsafe struct Node  
{  
 public int Value;  
 public Node\* Left;  
 public Node\* Right;  
}

модификатор unsafe, указанный в объявлении структуры, делает все текстовое пространство объявления структуры небезопасным контекстом. Таким образом можно объявить поля Left и Right с типом указателя. Этот пример можно было бы записать так:

public struct Node  
{  
 public int Value;  
 public unsafe Node\* Left;  
 public unsafe Node\* Right;  
}

Здесь модификаторы unsafe в объявлениях полей делают эти объявления небезопасными контекстами.

Кроме установления небезопасного контекста, дающего возможность использования типов указателей, модификатор unsafe не влияет на тип или член. В этом примере

public class A  
{  
 public unsafe virtual void F() {  
 char\* p;  
 ...  
 }  
}

public class B: A  
{  
 public override void F() {  
 base.F();  
 ...  
 }  
}

Модификатор unsafe метода F в классе A просто делает текстовое пространство F небезопасным контекстом, в котором можно использовать небезопасные возможности языка. В переопределении F в классе B нет необходимости вновь задавать модификатор unsafe, если, конечно, в самом методе F в классе B не требуются небезопасные возможности.

Ситуация несколько другая, когда тип указателя является частью подписи метода

public unsafe class A  
{  
 public virtual void F(char\* p) {...}  
}

public class B: A  
{  
 public unsafe override void F(char\* p) {...}  
}

Здесь, поскольку сигнатура F включает тип указателя, его можно записать только в небезопасном контексте. Однако небезопасный контекст можно ввести либо сделав небезопасным весь класс, как в случае A, либо включив модификатор unsafe в объявление метода, как в случае B.

## Типы указателей

В небезопасном контексте тип (§4) может быть типом\_указателя, а также типом\_значений или ссылочным\_типом. Однако *тип\_указателя* можно также использовать в выражении typeof (§7.6.10.6) вне небезопасного контекста, так как такое использование не является небезопасным.

type:  
...  
pointer-type

Тип\_указателя записывается как неуправляемый\_тип или как зарезервированное слово void с последующей лексемой \*:

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

Тип, указанный перед \* в типе указателя, называется типом\_референта типа указателя. Он представляет тип переменной, на которую указывает значение типа указателя.

В отличие от ссылок (значений ссылочных типов), указатели не отслеживаются сборщиком мусора—сборщику мусора неизвестны указатели и данные, на которые они указывают. По этой причине указателю не разрешено указывать на ссылку или структуру, содержащую ссылки, а тип референта указателя должен быть неуправляемого\_типа.

Неуправляемый\_тип — это любой тип, не являющийся ссылочным\_типом или сформированным типом и не содержащий поля ссылочного\_типа или сформированного типа на любом уровне вложенности. Иначе говоря, неуправляемым\_типом является один из следующих типов:

* sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal или bool.
* любой перечислимый\_тип;
* любой тип\_указателя;
* любой определенный пользователем тип\_структуры, который не является сформированным типом и содержащий поля только неуправляемых\_типов.

Интуитивное правило сочетания указателей и ссылок: референты ссылок (объекты) могут содержать указатели, но референты указателей не могут содержать ссылки.

Примеры типов указателей приведены в следующей таблице:

|  |  |
| --- | --- |
| **Пример** | **Описание** |
| byte\* | Указатель на byte |
| char\* | Указатель на char |
| int\*\* | Указатель на указатель на int |
| int\*[] | Одномерный массив указателей на int |
| void\* | указатель на неизвестный тип |

В данной реализации все типы указателей должны иметь одинаковый размер и представление.

В отличие от C и C++, если несколько указателей объявлены в одном объявлении, в C# звездочка \* записывается только вместе с базовым типом, а не как префиксный знак пунктуации с каждым именем указателя. Пример

int\* pi, pj; // NOT as int \*pi, \*pj;

Значение указателя, имеющего тип T\* представляет адрес переменной типа T. Оператор косвенного обращения к указателю \* (§18.5.1) можно использовать для доступа к этой переменной. Например, если дана

переменная P типа int\*, выражение \*P означает переменную типа int, находящуюся по адресу, который содержится в P.

Как и ссылка на объект, указатель может иметь значение null. Результат применения оператора косвенного обращения к указателю со значением null зависит от реализации. Указатель со значением null представляется нулями во всех разрядах.

Тип void\* представляет указатель на неизвестный тип. Поскольку тип референта неизвестен, оператор косвенного обращения нельзя применить к указателю типа void\*, и с таким указателем нельзя выполнять никакие вычисления. Однако указатель типа void\* можно привести к любому другому типу указателя (и наоборот).

Типы указателя являются отдельной категорией типов. В отличие от ссылочных типов и типов значения, типы указателя не наследуют от object и нет никаких преобразований между типами указателя и object. В частности, упаковка и распаковка (§4.3) не поддерживаются для указателей. Однако допускаются преобразования между различными типами указателя и между типами указателя и целыми типами. Это описано в §18.4.

Тип\_указателя не может использоваться в качестве аргумента типа (§4.4), а вывод типа (§7.5.2) не работает при вызове универсальных методов, в процессе которого в качестве аргумента типа будет выведен тип указателя.

Тип\_указателя можно использовать в качестве типа поля с модификатором volatile (§10.5.3).

Хотя указатели можно передавать в качестве параметров ref или out, это может привести к неопределенному поведению, поскольку указатель может быть установлен для указания на локальную переменную, которая больше не существует при возврате из вызванного метода, или если фиксированный объект, на который указывал указатель, больше не является фиксированным. Пример:

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  
 }  
 }  
}

Метод может возвращать значение некоторого типа, и этот тип может быть указателем. Например, если задан указатель на непрерывную последовательность элементов типа int, счетчика этой последовательности элементов и некоторого другого значения типа int, следующий метод возвратит адрес этого значения в этой последовательности, если есть соответствие, иначе возвращается null:

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;  
}

В небезопасном контексте имеется несколько конструкций для операций с указателями:

* оператор \* можно использовать для косвенного обращения к указателю (§18.5.1);
* оператор -> можно использовать для доступа к члену структуры посредством указателя (§18.5.2);
* оператор [] можно использовать для индексирования указателя (§18.5.3);
* оператор & можно использовать для получения адреса переменной (§18.5.4);
* операторы +++ и -- можно использовать для увеличения и уменьшения указателей (§18.5.5);
* операторы + и - можно использовать для выполнения арифметических операций с указателем (§18.5.6);
* операторы ==, !=, <, >, <=, and => можно использовать для сравнения указателей (§18.5.7).
* оператор stackalloc можно использовать для выделения памяти из стека вызовов (§18.7);
* оператор fixed можно использовать для временной фиксации переменной, чтобы можно было получить ее адрес (§18.6).

## Фиксированные и перемещаемые переменные

Оператор взятия адреса (§18.5.4) и оператор fixed (§18.6) разделяют переменные на две категории: фиксированные переменные и перемещаемые переменные.

Фиксированные переменные находятся в местах хранения, не затрагиваемых действием сборщика мусора. (Примерами фиксированных переменных являются локальные переменные, параметры по значению и переменные, созданные разыменованием указателей.) А перемещаемые переменные находятся в местах хранения, которые могут быть перемещены или удалены сборщиком мусора. (Примерами перемещаемых переменных являются поля в объектах и элементы массивов.)

Оператор & (§18.5.4) дает возможность без ограничений получать адрес фиксированной переменной. Но поскольку перемещаемая переменная может быть перемещена или удалена сборщиком мусора, адрес перемещаемой переменной можно получить только с помощью оператора fixed (§18.6), и этот адрес остается действительным только на время действия этого оператора fixed.

Точнее говоря, фиксированной переменной является одно из следующего:

* переменная, полученная из простого\_имени (§7.6.2), относящегося к локальной переменной или параметру по значению, если только эта переменная не захвачена анонимной функцией;
* переменная, полученная в результате доступа\_к\_члену (§7.6.4) вида V.I, где V — фиксированная переменная типа\_структуры;
* переменная, полученная в результате выражения\_косвенного\_обращения\_по\_указателю (§18.5.1) вида \*P, доступа\_к\_члену\_по\_указателю (§18.5.2) вида P->I или доступа\_к\_элементу\_по\_указателю (§18.5.3) вида P[E].

Все другие переменные классифицируются как перемещаемые.

Обратите внимание, что статическое поле классифицируется как перемещаемая переменная. Также обратите внимание, что параметр ref или out классифицируется как перемещаемая переменная, даже если аргумент, предоставленный для параметра, является фиксированной переменной. И наконец, обратите внимание, что переменная, созданная разыменованием указателя, всегда классифицируется как фиксированная переменная.

## Преобразования указателей

В небезопасном контексте набор доступных неявных преобразований (§6.1) расширен за счет включения следующих неявных преобразований указателей:

* от любого типа\_указателя к типу void\*.
* от литерала null к любому типу\_указателя.

Кроме того, в небезопасном контексте набор доступных явных преобразований (§6.2) расширен за счет включения следующих явных преобразований указателей:

* от любого типа\_указателя к любому другому типу\_указателя;
* Из типа sbyte, byte, short, ushort, int, uint, long или ulong в любой тип указателя.
* Из любого типа указателя в тип sbyte, byte, short, ushort, int, uint, long или ulong.

И наконец, в небезопасном контексте набор стандартных неявных преобразований (§6.3.1) включает следующее преобразование указателя:

* от любого типа\_указателя к типу void\*.

Преобразования между двумя типами указателей никогда не изменяют фактическое значение указателя. Иначе говоря, преобразование от одного типа указателя к другому не влияет на основной адрес, задаваемый указателем.

При преобразовании одного типа указателя к другому, если полученный указатель неправильно выровнен для указываемого типа, поведение является неопределенным, если результат разыменован. В общем, понятие "правильно выровненный" является транзитивным: если указатель на тип A правильно выровнен для указателя на тип B, который в свою очередь правильно выровнен для указателя на тип C, то указатель на тип A правильно выровнен для указателя на тип C.

Рассмотрите следующий случай, когда доступ к переменной одного типа выполняется через указатель на другой тип:

char c = 'A';  
char\* pc = &c;  
void\* pv = pc;  
int\* pi = (int\*)pv;  
int i = \*pi; // undefined  
\*pi = 123456; // undefined

При преобразовании типа указателя к указателю на байт результат указывает на младший адресуемый байт переменной. Последовательные приращения результата до размера переменной дают указатели на остальные байты этой переменной. Например, следующий метод отображает каждый из восьми байт переменной типа double в виде шестнадцатеричного значения:

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();  
 }  
 }  
}

Конечно, производимый вывод зависит от порядка следования байтов.

Сопоставления между указателями и целыми определяются реализацией. Однако на архитектурах 32- и 64-разрядных ЦП с линейным адресным пространством преобразования указателей к целым типам и целых типов к указателям обычно происходит точно так же, как преобразования значений типа uint или ulong соответственно к этим целым типам или от них к указателям.

### Массивы указателей

В небезопасном контексте можно формировать массивы указателей. В массивах указателей разрешены лишь некоторые преобразования, применимые к другим типам массива:

* Неявное преобразование ссылочного типа (§6.1.6) из любого типа\_массива в System.Array и реализуемые им интерфейсы применимо и к массивам указателей. Однако любая попытка получить доступ к элементам массива с помощью System.Array или реализуемых им интерфейсов приведет к исключению времени выполнения, так как типы указателей не могут быть преобразованы в object.
* Явные и неявные преобразования ссылочных типов (§6.1.6, §6.2.4) из типа одномерных массивов S[] в System.Collections.Generic.IList<T> и его универсальные базовые интерфейсы неприменимы к массивам указателей, поскольку типы указателей не могут использоваться в качестве аргументов типа, а преобразования из типов указателей в типы, не являющиеся типами указателей, невозможны.
* Явное преобразование ссылочного типа (§6.2.4) из System.Array и реализуемых им интерфейсов в любой тип\_массива применимо к массивам указателей.
* Явное и неявное преобразование ссылочных типов (§6.2.4) из System.Collections.Generic.IList<S> и его базовых интерфейсов в тип одномерных массивов T[] не применимо к массивам указателей, поскольку типы указателей не могут использоваться в качестве аргументов типа, а преобразования из типов указателей в типы, не являющиеся типами указателей, невозможны.

Эти ограничения означают, что расширение оператора foreach на массивы, описанные в §8.8.4, не применимы к массивам указателей. Вместо этого оператор foreach, заданный в виде

foreach (V v in x) внедренный оператор

, где x имеет тип массива вида T[,,…,], n — число измерений минус 1, а T или V имеют тип указателя, развернут с помощью вложенных циклов оператора for:

{  
 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);  
 внедренный\_оператор  
 }  
}

Переменные a, i0, i1, … in невидимы или недоступны для x, внедренного\_оператора или любого другого исходного кода программы. Переменная v доступна только для чтения во внедренном операторе. Если не определено явное преобразование (§18.4) типа T (типа элементов) в V, возникает ошибка, и больше никакие действия не выполняются. Если значение x равно null, во время выполнения возникает исключение System.NullReferenceException.

## Указатели в выражениях

В небезопасном контексте выражение может давать результат типа указателя, но вне небезопасного контекста выражение с типом указателя вызывает ошибку времени компиляции. Говоря точно, ошибка времени компиляции вне небезопасного контекста вызывается в случае, когда любое простое\_имя (§7.6.2), доступ\_к\_члену (§7.6.4), выражение\_вызова (§7.6.5) или доступ\_к\_элементу (§7.6.6) имеет тип указателя.

В небезопасном контексте порождения первичного\_выражения\_создания\_не\_массива (§7.6) и унарного\_выражения (§7.7) дают возможность создавать следующие дополнительные конструкции:

primary-no-array-creation-expression:  
...  
pointer-member-access  
pointer-element-access  
sizeof-expression

unary-expression:  
...  
pointer-indirection-expression  
addressof-expression

Эти конструкции описаны в следующих разделах. Приоритет и ассоциативность небезопасных операторов подразумеваются грамматикой.

### Косвенное обращение по указателю

Выражение\_косвенного\_обращения\_по\_указателю состоит из звездочки (\*) с последующим унарным\_выражением.

pointer-indirection-expression:  
\* unary-expression

Унарный оператор \* означает косвенное обращение по указателю и используется для получения переменной, на которую указывает указатель. Результатом вычисления \*P, где P — это выражение типа указателя T\*, является переменная типа T. Применение унарного оператора \* к выражению типа void\* или к выражению не типа указателя является ошибкой времени компиляции.

Результат применения унарного оператора \* к указателю со значением null определяется реализацией. В частности нет гарантии, что для этой операции будет создано исключение System.NullReferenceException.

Если указателю присвоено недопустимое значение, поведение унарного оператора \* является неопределенным. Среди недопустимых значений для разыменования указателя унарным оператором \* неправильно выровненный адрес указываемого типа (см. пример в §18.4) и адрес переменной после окончания ее времени жизни.

Для целей анализа определенного присваивания переменная, полученная вычислением выражения вида \*P, считается изначально присвоенной (§5.3.1).

### Доступ к члену по указателю

Доступ\_к\_члену\_по\_указателю состоит из первичного\_выражения, за которым следует лексема "->", а затем идентификатор и необязательный список\_аргументов\_типа.

pointer-member-access:  
primary-expression -> identifier type-argument-listopt

В доступе к члену по указателю вида P->I, P должно быть выражением типа указателя, отличного от void\*, а I должно обозначать доступный член того типа, на который указывает P.

Доступ к члену по указателю вида P->I вычисляется точно так же, как (\*P).I. Описание оператора косвенного обращения по указателю (\*) см. в §18.5.1. Описание оператора доступа к члену (.) см. в §7.6.4.

В этом примере

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());  
 }  
 }  
}

оператор -> используется для доступа к полям и вызова метода структуры с помощью указателя. Поскольку операция P->I совершенно эквивалентна (\*P).I, метод Main можно было записать так:

class Test  
{  
 static void Main() {  
 Point point;  
 unsafe {  
 Point\* p = &point;  
 (\*p).x = 10;  
 (\*p).y = 20;  
 Console.WriteLine((\*p).ToString());  
 }  
 }  
}

### Доступ к элементу по указателю

Доступ\_к\_элементу\_по\_указателю состоит из первичного\_выражения\_создания\_не\_массива, за которым следует выражение, заключенное в скобки "[" и "]".

pointer-element-access:  
primary-no-array-creation-expression [ expression ]

В выражении доступа к элементу по указателю P[E], P должно быть выражением типа указателя, отличного от void\*, а E должно быть выражением, которое может быть неявно преобразовано к типу int, uint, long или ulong.

Доступ к элементу по указателю вида P[E] вычисляется точно так же, как \*(P + E). Описание оператора косвенного обращения по указателю (\*) см. в §18.5.1. Описание оператора сложения указателей (+) см. в §18.5.6.

В этом примере

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) p[i] = (char)i;  
 }  
 }  
}

доступ к элементу по указателю используется для инициализации символьного буфера в цикле for. Поскольку операция P[E] совершенно эквивалентна \*(P + E), этот пример можно было записать так:

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) \*(p + i) = (char)i;  
 }  
 }  
}

Оператор доступа к элементу по указателю не проверяет на ошибки выхода за допустимые границы, и поведение при доступе к элементу вне границ является неопределенным. Это так же, как в C и C++.

### Оператор адреса

Выражение\_addressof состоит из амперсанда (&), за которым следует унарное\_выражение.

addressof-expression:  
& unary-expression

Если дано выражение E типа T и классифицируемое как фиксированная переменная (§18.3), конструкция &E вычисляет адрес переменной, заданной выражением E. Типом результата является T\*, классифицируемый как значение. Вызывается ошибка времени компиляции, если E не классифицируется как переменная, если E классифицируется как локальная переменная только для чтения или если E обозначает перемещаемую переменную. В этом последнем случае можно использовать оператор fixed (§18.6), чтобы временно "фиксировать" переменную перед получением ее адреса. Как изложено в §7.6.4, вне конструктора экземпляров или статического конструктора для структуры или класса, определяющего поле readonly, это поле считается значением, а не переменной. Его адрес, как таковой, нельзя получить. Аналогично и адрес константы получить нельзя.

Оператор & не требует, чтобы его аргументы были определенно присвоены, но после операции & та переменная, к которой применен этот оператор, считается определенно присвоенной на пути выполнения, на котором происходит эта операция. На программиста возлагается обязанность обеспечить, чтобы правильная инициализация переменной фактически происходила в такой ситуации.

В этом примере

using System;

class Test  
{  
 static void Main() {  
 int i;  
 unsafe {  
 int\* p = &i;  
 \*p = 123;  
 }  
 Console.WriteLine(i);  
 }  
}

i считается определенно присвоенным после операции &i, использованной для инициализации p. Присваивание для \*p в сущности инициализирует i, но включение этой инициализации является обязанностью программиста, и если удалить это присваивание, ошибки времени компиляции не будет.

Правила определенного присваивания для оператора & таковы, что можно избежать избыточной инициализации локальных переменных. Например, многие внешние API принимают указатель на структуру, заполненную посредством API. При вызове таких API обычно передается адрес локальной переменной типа структуры, и без этого правила потребовалась бы избыточная инициализация переменной типа структуры.

### Увеличение и уменьшение указателя

В небезопасном контексте операторы ++ и ‑‑ (§7.6.9 и §7.7.5) могут применяться к переменным типа указателя всех типов, кроме void\*. Так, для каждого типа указателя T\* неявно определены следующие операторы:

T\* operator ++(T\* x);

T\* operator --(T\* x);

Эти операторы дают такой же результат, как x + 1 и x - 1 соответственно (§18.5.6). Иначе говоря, для переменной типа указателя с типом T\* оператор ++ добавляет sizeof(T) к адресу, содержащемуся в переменной, а оператор ‑‑ вычитает sizeof(T) из адреса, содержащегося в переменной.

Если при операциях увеличения или уменьшения указателя переполняется область типа указателя, результат зависит от реализации, но исключения не создаются.

### Арифметические операции с указателем

В небезопасном контексте операторы + и - (§7.8.4 и §7.8.5) могут применяться к значениям всех типов указателей, кроме void\*. Так, для каждого типа указателя T\* неявно определены следующие операторы:

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);

Если дано выражение P типа указателя T\* и выражение N типа int, uint, long или ulong, то выражения P + N и N + P вычисляют значение указателя типа T\*, получающееся при добавлении значения N \* sizeof(T) к адресу, заданному выражением P. Аналогичным образом, выражение P - N вычисляет значение указателя типа T\*, получающееся при вычитании значения N \* sizeof(T) из адреса, заданного выражением P.

Если даны два выражения, P и Q, типа указателя T\*, то выражение P – Q вычисляет разность адресов, заданных P и Q, а затем делит эту разность на sizeof(T). Тип результата всегда long. В действительности P - Q вычисляется как ((long)(P) - (long)(Q)) / sizeof(T).

Пример:

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);  
 }  
 }  
}

В результате получается:

p - q = -14  
q - p = 14

Если при арифметических операциях с указателем переполняется область типа указателя, результат усекается способом, зависящим от реализации, но исключения не создаются.

### Сравнение указателей

В небезопасном контексте операторы ==, !=, <, >, <= и => (§7.10) можно применять к значениям всех типов указателей. Операторы сравнения указателей следующие:

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);

Поскольку существует неявное преобразование любого типа указателей к типу void\*, операнды любого типа указателей можно сравнивать с помощью этих операторов. Операторы сравнения сравнивают адреса, заданные двумя операндами, как если бы они были беззнаковыми целыми.

### Оператор sizeof

Оператор sizeof возвращает число байт, занимаемых переменной данного типа. Тип, указанный в качестве операнда sizeof, должен быть неуправляемым\_типом (§18.2).

sizeof-expression:  
sizeof ( unmanaged-type )

Результатом оператора sizeof является значение типа int. Для некоторых предопределенных типов оператор sizeof дает постоянное значение, как указано в приведенной ниже таблице.

|  |  |
| --- | --- |
| **Выражение** | **Результат** |
| sizeof(sbyte) | 1 |
| sizeof(byte) | 1 |
| sizeof(short) | 2 |
| sizeof(ushort) | 2 |
| sizeof(int) | 4 |
| sizeof(uint) | 4 |
| sizeof(long) | 8 |
| sizeof(ulong) | 8 |
| sizeof(char) | 2 |
| sizeof(float) | 4 |
| sizeof(double) | 8 |
| sizeof(bool) | 1 |

Для всех других типов результат оператора sizeof определяется реализацией и классифицируется как значение, а не константа.

Порядок объединения членов в структуру не установлен.

Для целей выравнивания могут быть безымянные заполнения в начале, внутри и в конце структуры. Содержимое разрядов, используемых в качестве заполнения, не определено.

При применении к операнду, имеющему тип структуры, результатом является общее число байт в переменной этого типа, включая заполнения.

## Оператор fixed

В небезопасном контексте порождение внедренного\_оператора (§8) допускает дополнительную конструкцию, оператор fixed, используемый для "фиксирования" перемещаемой переменной, так что ее адрес остается постоянным на время действия этого оператора.

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

Каждый декларатор\_указателя\_fixed объявляет локальную переменную данного типа\_указателя и инициализирует эту локальную переменную адресом, вычисленным соответствующим инициализатором\_указателя\_fixed. Локальная переменная, объявленная в операторе fixed, доступна в любом инициализаторе\_указателя\_fixed, находящемся справа от объявления этой переменной, и во внедренном\_операторе оператора fixed. Локальная переменная, объявленная оператором fixed, считается доступной только для чтения. Возникает ошибка времени компиляции, если внедренный оператор пытается изменить эту локальную переменную (через присваивание или операторами ++ и ‑‑) или передать ее в качестве параметра ref или out.

Инициализатором\_указателя\_fixed может быть одно из следующего:

* лексема "&", за которой следует ссылка\_на\_переменную (§5.3.3) на перемещаемую переменную (§18.3) неуправляемого типа T при условии, что тип T\* является неявно преобразуемым к типу указателя, заданному в операторе fixed. В этом случае инициализатор вычисляет адрес заданной переменной и эта переменная гарантированно остается по фиксированному адресу на время действия оператора fixed;
* выражение типа\_массива с элементами неуправляемого типа T при условии, что тип T\* является неявно преобразуемым к типу, заданному в операторе fixed. В этом случае инициализатор вычисляет адрес первого элемента массива, а весь массив гарантированно остается по фиксированному адресу на время действия оператора fixed. Поведение оператора fixed определяется реализацией, если выражение массива дает null или если в массиве нуль элементов;
* выражение типа string при условии, что тип char\* является неявно преобразуемым к типу указателя, заданному в операторе fixed. В этом случае инициализатор вычисляет адрес первого символа строки, а вся строка гарантированно остается по фиксированному адресу на время действия оператора fixed. Поведение оператора fixed определяется реализацией, если строковое выражение дает null;
* простое\_имя или доступ\_к\_члену, ссылающиеся на член буфера фиксированного объема перемещаемой переменной, при условии, что тип члена буфера фиксированного объема является неявно преобразуемым к типу указателя, заданному в операторе fixed. В этом случае инициализатор вычисляет указатель на первый элемент буфера фиксированного объема (§18.7.2), а буфер фиксированного объема гарантированно остается по фиксированному адресу на время действия оператора fixed.

Для каждого адреса, вычисленного инициализатором\_указателя\_fixed, оператор fixed гарантирует, что переменная, находящаяся по этому адресу, не подлежит перемещению или удалению сборщиком мусора на время действия оператора fixed. Например, если адрес, вычисленный инициализатором\_указателя\_fixed, ссылается на поле объекта или на элемент экземпляра массива, оператор fixed гарантирует, что содержащий экземпляр объекта не будет перемещен или удален в течение срока жизни этого оператора.

На программиста возлагается обязанность обеспечить, чтобы указатели, созданные операторами fixed, не продолжали существовать за пределами выполнения этих операторов. Например, если указатели, созданные операторами fixed, передаются внешним API, программист обязан обеспечить, чтобы в этих API не сохранялась память этих указателей.

Фиксированные объекты могут вызвать фрагментацию кучи (поскольку их нельзя перемещать). По этой причине объекты следует фиксировать только при абсолютной необходимости и на как можно более короткий срок.

Пример:

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);  
 }  
 }  
}

демонстрируется несколько способов использования оператора fixed. Первый оператор фиксирует и получает адрес статического поля, второй оператор фиксирует и получает адрес поля экземпляра, а третий оператор фиксирует и получает адрес элемента массива. В каждом случае было бы ошибкой использовать обычный оператор &, так как эти переменные классифицируются как перемещаемые переменные.

Четвертый оператор fixed в приведенном выше примере дает тот же результат, что и третий.

В этом примере оператора fixed используется тип string:

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);  
 }  
 }  
}

В небезопасном контексте элементы массива в одномерном массиве хранятся в порядке возрастания индекса, начиная с индекса 0 и заканчивая индексом Length – 1. В многомерных массивах элементы хранятся так, что сначала возрастают индексы самого правого измерения, затем соседнего левого и так далее влево. В пределах оператора fixed, получающего указатель p на экземпляр массива a, значения указателя в диапазоне от pдо p + a.Length - 1 представляют адреса элементов массива. Аналогично переменные в диапазоне от p[0] до p[a.Length - 1] представляют фактические элементы массива. При данном способе хранения массивов можно обращаться с массивом любого измерения как если бы он был линейным.

Пример:

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.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);  
 Console.WriteLine();  
 }  
 }  
}

В результате получается:

[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

В этом примере

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);  
 }  
 }  
}

оператор fixed используется для фиксирования массива, чтобы его адрес передать методу, принимающему указатель.

В примере:

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);  
 }  
 }  
}

оператор fixed используется, чтобы фиксировать буфер фиксированного размера структуры, так чтобы его адрес можно было использовать в качестве указателя.

Значение char\*, созданное фиксацией экземпляра строки, всегда указывает на строку, оканчивающуюся символом null. В пределах оператора fixed, получающего указатель p на экземпляр строки s, значения указателя в диапазоне от p до p + s.Length - 1 представляют адреса символов в строке, а значение указателя p + s.Length всегда указывает на символ null (символ со значением '\0').

Изменение объектов управляемого типа посредством фиксированных указателей может привести к неопределенному поведению. Например, поскольку строки являются неизменяемыми, на программиста возлагается обязанность обеспечить, чтобы символы, на которые ссылается указатель на фиксированную строку, не изменялись.

Автоматическое завершение строк символом null особенно удобно при вызове внешних API, ожидающих строки «в стиле C». Отметьте, однако, что в экземпляре строки могут содержаться символы null. Если символы null имеются, то строка будет выглядеть усеченной, если обращаться с ней как с завершающейся символом null строкой char\*.

## Буферы фиксированного размера

Буферы фиксированного размера используются для объявления «в стиле C» линейных массивов как членов структур, главным образом они полезны для связи с неуправляемыми API.

### Объявления буферов фиксированного размера

Буфер фиксированного размера является членом, представляющим хранилище для буфера фиксированной длины для переменных заданного типа. Объявление буфера фиксированного размера вводит один или более буферов фиксированного размера с заданным типом элементов. Буферы фиксированного размера допускаются только в объявлениях структур и могут быть только в небезопасных контекстах (§18.1).

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 ]

Объявление буфера фиксированного размера может включать набор атрибутов (§17), модификатор new (§10.2.2), допустимое сочетание из четырех модификаторов доступа (§10.2.3) и модификатор unsafe (§18.1). Атрибуты и модификаторы применяются ко всем членам, объявленным в объявлении буфера фиксированного размера. Является ошибкой, если один и тот же модификатор появляется несколько раз в объявлении буфера фиксированного размера.

В объявление буфера фиксированного размера нельзя включать модификатор static.

Тип элемента буфера в объявлении буфера фиксированного размера указывает тип элемента буферов, введенных объявлением. Тип элемента буфера должен быть одним из предопределенных типов sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double или bool.

За типом элемента буфера следует список деклараторов буфера фиксированного размера, каждый из которых вводит новый член. Декларатор буфера фиксированного размера состоит из идентификатора, именующего член, за ним следует константное выражение, заключенное в лексемы [ и ]. Константное выражение указывает число элементов в члене, введенном этим декларатором буфера фиксированного размера. Тип константного выражения должен быть неявно преобразуемым к типу int, а значение должно быть ненулевым положительным целым числом.

Элементы буфера фиксированного размера гарантированно размещаются последовательно в памяти.

Объявление буфера фиксированного размера, в котором объявлено несколько буферов фиксированного размера, эквивалентно нескольким объявлениям с объявлением одного буфера фиксированного размера с теми же атрибутами и типами элементов. Пример

unsafe struct A  
{  
 public fixed int x[5], y[10], z[100];  
}

равнозначно

unsafe struct A  
{  
 public fixed int x[5];  
 public fixed int y[10];  
 public fixed int z[100];  
}

### Буферы фиксированного размера в выражениях

Поиск члена (§7.3) буфера фиксированного размера выполняется точно так же, как поиск поля.

На буфер фиксированного размера можно ссылаться в выражении с помощью простого\_имени (§7.5.2) или доступа\_к\_члену (§7.5.4).

При ссылке на член буфера фиксированного размера по простому имени результат такой же, как при доступе к члену в виде this.I, где I — член буфера фиксированного размера.

При доступе к члену в виде E.I если E имеет тип структуры и поиск члена I в этом типе структуры идентифицирует член фиксированного размера, E.I вычисляется и классифицируется следующим образом:

* если выражение E.I встречается не в небезопасном контексте, выдается ошибка времени компиляции;
* если E классифицируется как значение, выдается ошибка времени компиляции;
* иначе, если E является перемещаемой переменной (§18.3), а выражение E.I не является инициализатором\_указателя\_fixed (§18.6), выдается ошибка времени компиляции;
* иначе E ссылается на фиксированную переменную, а результатом выражения является указатель на первый элемент члена I буфера фиксированного размера в E. Результат имеет тип S\*, где S — тип элемента I, и классифицируется как значение.

Доступ к последующим элементам буфера фиксированного размера можно получить с помощью операций с указателем от первого элемента. В отличие от доступа к массивам, доступ к элементам буфера фиксированного размера является небезопасной операцией без проверки диапазона.

В следующем примере объявляется и используется структура с членом, являющимся буфером фиксированного размера.

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);  
 }  
}

### Проверка определенного присваивания

Буферы фиксированного размера не подлежат проверке определенного присваивания (§5.3), а члены буфера фиксированного размера игнорируются при проверке определенного присваивания переменным типа структуры.

Если самая дальняя переменная типа структуры, включающая элемент буфера фиксированного размера, является статической переменной, переменной экземпляра класса или элементом массива, элементы буфера фиксированного размера автоматически инициализируются своими значениями по умолчанию (§5.2). Во всех других случаях начальное содержимое буфера фиксированного размера является неопределенным.

## Выделение стека

В небезопасном контексте объявление локальной переменной (§8.5.1) может включать инициализатор выделения стека, который выделяет память из стека вызова.

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

Неуправляемый\_тип указывает тип элементов, которые будут сохраняться во вновь выделенном месте, а выражение указывает количество этих элементов. Совместно они указывают необходимый размер выделения. Так как размер выделения стека не может быть отрицательным, выдается ошибка времени компиляции, если число элементов задано константным\_выражением, дающим в результате отрицательное значение.

Для инициализатора выделения стека в виде stackalloc T[E] требуется, чтобы T было неуправляемым типом (§18.2), а E — выражением типа int. Эта конструкция выделяет E \* sizeof(T) байт из стека вызова и возвращает указатель (типа T\*) на вновь выделенный блок. Если E — отрицательное значение, то поведение неопределенное. Если E равно нулю, то выделение не производится, а возвращаемый указатель определяется реализацией. Если не хватает памяти для выделения блока заданного размера, выдается System.StackOverflowException.

Содержимое вновь выделенной памяти неопределенное.

Инициализаторы выделения стека не разрешено использовать в блоках catch и finally (§8.10).

Не существует способа явного освобождения памяти, выделенной с помощью stackalloc. Все блоки выделенной стеку памяти, созданные во время выполнения члена функции, автоматически удаляются при возврате из члена функции. Это соответствует функции alloca, расширение, обычно встречающееся в реализациях C и C++.

В этом примере

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));  
 }  
}

инициализатор stackalloc используется в методе IntToString для выделения буфера размером 16 символов в стеке. Этот буфер автоматически удаляется при возврате метода.

## Динамическое выделение памяти

Кроме оператора stackalloc, C# не предоставляет предопределенные конструкции для управления памятью, не собираемой сборщиком мусора. Такие службы обычно предоставляются поддержкой библиотек классов или импортируются непосредственно из операционной системы. В следующем примере показано в классе Memory, как можно обратиться к функциям кучи операционной системы из C#:

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);  
}

Пример использования класса Memory:

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]);  
 }  
 }  
}

В этом примере выделяется 256 байт памяти через Memory.Alloc и блок памяти инициализируется значениями, возрастающими от 0 до 255. Затем размещается байтовый массив из 256 элементов и используется Memory.Copy для копирования содержимого блока памяти в байтовый массив. Наконец, этот блок памяти освобождается с помощью Memory.Free и содержимое байтового массива выводится на консоль.

1. Комментарии к документации

C# предоставляет программистам механизм документирования своего кода с помощью специального синтаксиса комментариев с XML-текстом. Комментарии в файлах исходного кода, имеющие определенный вид, могут быть использованы для управления инструментом создания XML из этих комментариев и элементов исходного кода, которым они предшествуют. Комментарии, использующие такой синтаксис, называются комментариями к документации. Они должны непосредственно предшествовать пользовательскому типу (такому как класс, делегат или интерфейс) или члену (такому как поле, событие, свойство или метод), который они комментируют. Средство создания XML называется генератором документации. (Этим генератором может быть сам компилятор C# или другое средство.) Создаваемый генератором документации вывод называется файлом документации. Файл документации используется в качестве входных данных для средства просмотра документации — инструмента для визуального представления сведений о типе и сопутствующей документации.

Эта спецификация предлагает набор тегов для использования в комментариях к документации, но использование этих тегов не является обязательным, можно при желании использовать другие теги, если соблюдаются правила правильного XML.

* 1. Введение

Комментарии, имеющие специальный вид, могут быть использованы для управления инструментом создания XML из этих комментариев и элементов исходного кода, которым они предшествуют. Такие комментарии являются однострочными комментариями, начинающимися с трех косых черт (///), или комментариями с разделителями, начинающимися с косой черты и двух звездочек (/\*\*). Они должны непосредственно предшествовать пользовательскому типу (такому как класс, делегат или интерфейс) или члену (такому как поле, событие, свойство или метод), который они комментируют. Разделы атрибутов (§17.2) считаются частью объявлений, так что комментарии к документации должны предшествовать атрибутам, примененным к типу или члену.

**Синтаксис:**

single-line-doc-comment:  
/// input-charactersopt

delimited-doc-comment:  
/\*\* delimited-comment-textopt \*/

Если в однострочном\_комментарии\_к\_документации символ пустого\_пространства следует за символами /// в каждом из однострочных\_комментариев\_к\_документации, примыкающих к текущему однострочному\_комментарию\_к\_документации, то этот символ пустого\_пространства не включается в XML-вывод.

Если в комментарии\_с\_разделителями\_к\_документации первый символ на второй строке, не являющийся символом пустого\_пространства, является звездочкой, и тот же порядок необязательных символов пустого\_пространства и символа звездочки повторяется в начале каждой строки внутри комментария\_с\_разделителями\_к\_документации, то эта повторяющаяся комбинация символов не включается в XML-вывод. Символы пустого\_пространства могут входить в эту комбинацию как до, так и после символа звездочки.

**Пример:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>  
///  
public class Point   
{  
 /// <summary>method <c>draw</c> renders the point.</summary>  
 void draw() {…}  
}

Текст в комментариях к документации должен быть правильным согласно правилам XML (http://www.w3.org/TR/REC-xml). При неправильном XML создается предупреждение и файл документации будет содержать комментарий, сообщающий об ошибке.

Хотя разработчики могут создавать свои собственные наборы тегов, рекомендуемый набор определен в §A.2. Некоторые из рекомендуемых тегов имеют специальные значения.

* Тег <param> используется для описания параметров. Если используется этот тег, генератор документации должен проверить, что указанный параметр существует и что все параметры описаны в комментариях к документации. Если проверка неуспешна, генератор документации выдает предупреждение;
* Атрибут cref может быть присоединен к любому тегу для предоставления ссылки на элемент кода. Генератор документации должен проверить, что этот элемент кода существует. Если проверка неуспешна, генератор документации выдает предупреждение. При поиске имени, описанного в атрибуте cref, генератор документации должен принимать во внимание видимость пространства имен в соответствии с операторами using, встречающимися в исходном коде. Для общих элементов кода нельзя использовать обычный универсальный синтаксис (т. е. "List<T>"), так как он создает неправильный XML. Можно использовать фигурные скобки вместо угловых (т. е. List{T}) или escape-синтаксис XML (т. е. List&lt;T&gt;):
* тег <summary> предназначен для использования средством просмотра документации для отображения дополнительных сведений о типе или члене;
* тег <include> включает сведения из внешнего XML-файла.

Внимательно отнеситесь к тому, что файл документации не предоставляет полные сведения о типе и членах (например, он не содержит никакие сведения о типе). Для получения таких сведений о типе или члене файл документации необходимо использовать в сочетании с соображениями по фактическому типу или члену.

* 1. Рекомендованные теги

Генератор документации обязан принять и обработать любой тег, допустимый по правилам XML. Следующие теги предоставляют обычно используемую функциональность в пользовательской документации. (Конечно, возможны и другие теги.)

|  |  |  |
| --- | --- | --- |
| **Тег** | **Раздел** | **Назначение** |
| <c> | A.2.1 | Установить шрифт текста, подобный исходному коду |
| <code> | A.2.2 | Установить одну или несколько строк исходного кода или программного вывода |
| <example> | A.2.3 | Указать пример |
| <exception> | A.2.4 | Идентификация исключений, которые могут выдаваться методом |
| <include> | A.2.5 | Включение XML из внешнего файла |
| <list> | A.2.6 | Создание списка или таблицы |
| <para> | A.2.7 | Разрешить добавление структуры к тексту |
| <param> | A.2.8 | Описать параметр для метода или конструктора |
| <paramref> | A.2.9 | Указать, что слово является именем параметра |
| <permission> | A.2.10 | Документировать специальные возможности безопасности члена |
| <remark> | A.2.11 | Предоставление дополнительных сведений о типе |
| <returns> | A.2.12 | Описать возвращаемое методом значение |
| <see> | A.2.13 | Указать ссылку |
| <seealso> | A.2.14 | Создать запись См. также |
| <summary> | A.2.15 | Описание типа или члена типа |
| <value> | A.2.16 | Описать свойство |
| <typeparam> |  | Описать параметр универсального типа |
| <typeparamref> |  | Указать, что слово является именем параметра типа |

* + 1. <c>

Этот тег предоставляет механизм для указания, что для фрагмента текста внутри описания следует установить особый шрифт, такой как используемый для блока кода. Для строк фактического текста используйте тег <code> (§A.2.2).

**Синтаксис:**

<c>*text*</c>

**Пример:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>

public class Point   
{  
 // ...  
}

* + 1. <code>

Этот тег используется для установки для одной или нескольких строк исходного кода или программного вывода особого шрифта. Для небольших фрагментов кода в комментарии используйте <c> (§A.2.1).

**Синтаксис:**

<code>*source code or program output*</code>

**Пример:**

/// <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;  
}

* + 1. <example>

Этот тег позволяет вставить пример кода в комментарий, чтобы указать, как можно использовать метод или другой член библиотеки. Обычно это влечет также и использование тега <code> (§A.2.2).

**Синтаксис:**

<example>*description*</example>

**Пример:**

Пример см. в разделе <code> (§A.2.2).

* + 1. <exception>

Этот тег предоставляет способ документирования исключений, которые может выдавать метод.

**Синтаксис:**

<exception cref="*member*">*description*</exception>

где

cref="*member*"

имя члена. Генератор документации проверяет, что данный член существует, и преобразует член в каноническое имя элемента в файле документации.

*description*

Описание обстоятельств, при которых выдается это исключение.

**Пример:**

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();  
 // …  
 }   
}

* + 1. <include>

Этот тег позволяет включать сведения из внешнего XML-документа в файл исходного кода. Внешний файл должен быть правильным XML-документом, а выражение XPath применяется к этому документу для указания, какой XML из этого документа требуется включить. Затем тег <include> заменяется выбранным XML из внешнего документа.

**Синтаксис:**

<include file="filename" path="xpath" />

где

file="filename"

Имя внешнего XML-файла. Имя файла интерпретируется относительно файла, содержащего тег включения.

path="xpath"

Выражение Xpath, выбирающее один из XML во внешнем XML-файле.

**Пример:**

Если в исходном коде содержится такое объявление:

/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/\*' />  
public class IntList { … }

а во внешнем файле "docs.xml" есть следующее содержимое:

<?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>

тогда эта же документация выводится, как если бы исходный код содержал:

/// <summary>  
/// Contains a list of integers.  
/// </summary>  
public class IntList { … }

* + 1. <list>

Этот тег используется для создания списка или таблицы элементов. Он может содержать блок <listheader> для определения строки заголовка таблицы или списка определений. (При определении таблицы необходимо предоставить только запись *term* в заголовке.)

Каждый элемент списка задается блоком <item>. При создании списка определений необходимо одновременно задать элементы *term* и *description*. Но для таблицы, маркированного списка или нумерованного списка необходимо задать только *description*.

**Синтаксис:**

<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>

где

*term*

Определяемый термин, заданный в *description*.

*description*

либо элемент маркированного или нумерованного списка, либо определение *term*.

**Пример:**

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 () {  
 // ...  
 }  
}

* + 1. <para>

Этот тег используется внутри других тегов, таких как <summary> (§A.2.11) или <returns> (§A.2.12), и позволяет добавить структуру к тексту.

**Синтаксис:**

<para>*content*</para>

где

*content*

текст абзаца.

**Пример:**

/// <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-trvial maintenance has  
/// been performed on the Point class.</para></summary>  
public static void Main() {  
 // ...  
}

* + 1. <param>

Этот тег используется для описания параметра метода, конструктора или индексатора.

**Синтаксис:**

<param name="*name*">*description*</param>

где

*name*

имя параметра.

*description*

описание параметра.

**Пример:**

/// <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;  
}

* + 1. <paramref>

Этот тег используется для указания, что слово является параметром. Файл документации может быть обработан так, чтобы этот параметр был представлен выделяющимся образом.

**Синтаксис:**

<paramref name="*name*"/>

где

*name*

имя параметра.

**Пример:**

/// <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;  
}

* + 1. <permission>

Этот тег дает возможность документировать специальные возможности безопасности члена.

**Синтаксис:**

<permission cref="*member*">*description*</permission>

где

cref="*member*"

имя члена. Генератор документации проверяет, что данный элемент кода существует, и преобразует член в каноническое имя элемента в файле документации.

*description*

описание доступа к члену.

**Пример:**

/// <permission cref="System.Security.PermissionSet">Everyone can  
/// access this method.</permission>

public static void Test() {  
 // ...  
}

* + 1. <remark>

Этот тег используется для указания дополнительных сведений о типе. (Используйте <summary> (§A.2.15) для описания самого типа и членов типа.)

**Синтаксис:**

<remark>*description*</remark>

где

*description*

Текст примечания.

**Пример:**

/// <summary>Class <c>Point</c> models a point in a   
/// two-dimensional plane.</summary>  
/// <remark>Uses polar coordinates</remark>  
public class Point   
{  
 // ...  
}

* + 1. <returns>

Этот тег используется для описания возвращаемого методом значения.

**Синтаксис:**

<returns>*description*</returns>

где

*description*

описание возвращаемого значения.

**Пример:**

/// <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 + ")";  
}

* + 1. <see>

Этот тег дает возможность указать ссылку внутри текста. Используйте <seealso> (§A.2.14) для указания текста, который должен быть представлен в разделе See Also.

**Синтаксис:**

<see cref="*member*"/>

где

cref="*member*"

имя члена. Генератор документации проверяет, что данный элемент кода существует, и заменяет член на имя элемента в сгенерированном файле документации.

**Пример:**

/// <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;  
}

* + 1. <seealso>

Этот тег дает возможность создания записи для раздела См. также. Используйте <see> (§A.2.13) для указания ссылки изнутри текста.

**Синтаксис:**

<seealso cref="*member*"/>

где

cref="*member*"

имя члена. Генератор документации проверяет, что данный элемент кода существует, и заменяет член на имя элемента в сгенерированном файле документации.

**Пример:**

/// <summary>This method determines whether two Points have the same  
/// location.</summary>  
/// <seealso cref="operator=="/>  
/// <seealso cref="operator!="/>  
public override bool Equals(object o) {  
 // ...  
}

* + 1. <summary>

Этот тег используется для описания типа или члена типа. Используйте <remark> (§A.2.11) для описания самого типа.

**Синтаксис:**

<summary>*description*</summary>

где

*description*

Сводные сведения о типе или члене.

**Пример:**

/// <summary>This constructor initializes the new Point to (0,0).</summary>  
public Point() : this(0,0) {  
}

* + 1. <value>

Этот тег дает возможность описать свойство.

**Синтаксис:**

<value>*property* *description*</value>

где

*property description*

описание свойства.

**Пример:**

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
public int X  
{  
 get { return x; }  
 set { x = value; }  
}

* + 1. <typeparam>

Этот тег используется для описания параметра универсального типа для класса, структуры, интерфейса, делегата и метода.

**Синтаксис:**

<typeparam name="*name*">*description*</typeparam>

где

*name*

имя параметра типа.

*description*

описание параметра типа.

**Пример:**

/// <summary>A generic list class.</summary>  
/// <typeparam name="T">The type stored by the list.</typeparam>  
public class MyList<T> {  
 ...  
}

* + 1. <typeparamref>

Этот тег используется для указания, что слово является параметром типа. Файл документации может быть обработан так, чтобы этот параметр типа был представлен выделяющимся образом.

**Синтаксис:**

<typeparamref name="*name*"/>

где

*name*

имя параметра типа.

**Пример:**

/// <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) {  
 ...  
}

* 1. Обработка файла документации

Генератор документации создает строку идентификатора для каждого элемента исходного кода, помеченного комментарием к документации. Эта строка идентификатора однозначно идентифицирует элемент источника. Средство просмотра документации может использовать строку идентификатора для идентификации соответствующего элемента метаданных или отражения, к которому применяется документация.

Файл документации не является иерархическим представлением исходного кода, это, скорее, единообразный список с созданной для каждого элемента строкой идентификатора.

* + 1. Формат строки идентификатора

При создании строк идентификаторов генератор документации соблюдает следующие правила:

* в строку не заносятся пробелы;
* первая часть строки идентифицирует вид документируемого члена с помощью одного символа, за которым следует двоеточие. Определены следующие виды членов:

|  |  |
| --- | --- |
| **Символ** | **Описание** |
| E | Событие |
| F | Поле |
| M | Метод (включая конструкторы, деструкторы и операторы) |
| N | Пространство имен |
| P | Свойство (включая индексаторы) |
| T | Тип (такой как класс, делегат, перечисление, интерфейс или структура) |
| ! | Строка ошибки; в остальной части строки приведены сведения об ошибке. Например, генератор документации создает сведения об ошибке для ссылок, которые не удается разрешить. |

* во второй части строки находится полное имя элемента, начиная от корня пространства имен. Имена элемента, вмещающих его типов и пространства имен разделены точками. Если в самом имени есть точки, они заменяются символами # (U+0023). (Предполагается, что в именах элементов нет такого символа);
* для методов и свойств с аргументами следует список аргументов, заключенный в круглые скобки. Если аргументов нет, скобки опускаются. Аргументы разделяются запятыми. Кодирование каждого аргумента такое же, как у подписи CLI, заключается в следующем:
* аргументы представлены своими именами в документации, основанными на их полном имени, со следующими изменениями:

к аргументам, представляющим универсальные типы, присоединяется символ «’», за которым следует число параметров типа;

у аргументов с модификатором out или ref за именем типа следует @. Аргументы, передаваемые по значению или через params, не имеют специального обозначения;

Аргументы, являющиеся массивами, представляются как [ нижняя\_граница : размер , … , нижняя\_граница : размер ], где число запятых равно рангу минус один, а нижние границы и размер каждого измерения, если они известны, представляются десятичными числами. Если нижняя граница или размер не указаны, они опускаются. Если для отдельного измерения опущены и нижняя граница, и размер, то ":" тоже опускается. Массивы массивов представлены одним "[]" на каждый уровень:

аргументы, имеющие типы указателей, отличные от void, представляются с помощью \* вслед за именем типа. Указатель типа void представляется с помощью имени типа System.Void;

аргументы, ссылающиеся на параметры универсального типа, определенные для типов, кодируются с помощью символа «`», за которым следует начинающийся с нуля индекс параметра типа;

аргументы, использующие параметры универсального типа, определенные в методах, используют двойной обратный апостроф «``» вместо одинарного «`», используемого для типов;

аргументы, ссылающиеся на построенные универсальные типы, кодируются с помощью универсального типа, за которым следует «{», затем разделенный запятыми список аргументов типа и затем «}».

* + 1. Примеры строк идентификаторов

В каждом из следующих примеров показан фрагмент кода на C# вместе со строками идентификаторов, созданных по каждому исходному элементу, для которого возможен комментарий к документации:

* типы представлены своими полными именами, дополненными общими сведениями:

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”

* поля представлены своими полными именами:

namespace Acme  
{  
 struct ValueType  
 {  
 private int total;  
 }

class Widget: IProcess  
 {  
 public class NestedClass  
 {  
 private int value;  
 }

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"

* конструкторы;

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)"

* деструкторы;

namespace Acme  
{  
 class Widget: IProcess  
 {  
 ~Widget() {...}  
 }  
}

"M:Acme.Widget.Finalize"

* методы;

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)”

* свойства и индексаторы;

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)"

* события;

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public event Del AnEvent;  
 }  
}

"E:Acme.Widget.AnEvent"

* Унарные операторы.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x) {...}  
 }  
}

"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"

Полный набор имен функций унарных операторов: op\_UnaryPlus, op\_UnaryNegation, op\_LogicalNot, op\_OnesComplement, op\_Increment, op\_Decrement, op\_True и op\_False.

* Бинарные операторы.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x1, Widget x2) {...}  
 }  
}

"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

Полный набор имен функций бинарных операторов: 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 и op\_GreaterThanOrEqual.

* операторы преобразования оканчиваются символом "~", за которым следует тип возвращаемого значения;

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"

* 1. Пример
     1. Исходный код C#

В следующем примере показан исходный код класса Point:

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> is the new Point's x-coordinate.</param>  
 /// <param><c>yor</c> 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> is the new x-coordinate.</param>  
 /// <param><c>yor</c> 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> is the relative x-offset.</param>  
 /// <param><c>yor</c> 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> 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, training, 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> 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="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> 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="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-trvial maintenance has  
 /// been performed on the Point class.</para></summary>  
 public static void Main() {  
 // class test code goes here  
 }  
}  
}

* + 1. Результирующий XML

Здесь представлен результат, созданный генератором документации по вышеприведенному исходному коду для класса Point:

<?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>

<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, training, 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-trvial 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>

1. Грамматика

Это приложение содержит сводные сведения о лексике и синтаксисе, используемых в основном документе, а также о грамматических расширениях для небезопасного кода. Грамматические конструкции отображаются здесь в том же порядке, в котором они представлены в основном документе.

* 1. Лексика

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

* + 1. Знаки завершения строки

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)

* + 1. Комментарии

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 \*

* + 1. Пробел

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

* + 1. Лексемы

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

* + 1. Управляющие последовательности символов Юникода

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 hex-digit

* + 1. Идентификаторы

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

* + 1. Ключевые слова

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

* + 1. Литералы

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-suffixopt

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 uL ul 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

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt 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-digitopt hex-digitopt hex-digitopt

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

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:  
""

null-literal:  
null

* + 1. Операторы и знаки пунктуации

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

right-shift:  
>|>

right-shift-assignment:  
>|>=

* + 1. Директивы предварительной обработки

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:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

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

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

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

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 #

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

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

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

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 "

pp-pragma:  
whitespaceopt # whitespaceopt 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 whitespaceopt , whitespaceopt decimal-digits

* 1. Синтаксис
     1. Основные понятия

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-listoptqualified-alias-member

* + 1. Типы

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

* + 1. Переменные

variable-reference:  
expression

* + 1. Выражения

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt 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-listopt

parenthesized-expression:  
( expression )

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
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-listopt )

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-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

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

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

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-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

delegate-creation-expression:  
new delegate-type ( expression )

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  
identifier = expression

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 ,

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

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-signatureopt block

anonymous-function-signature:  
explicit-anonymous-function-signature   
implicit-anonymous-function-signature

explicit-anonymous-function-signature:  
( explicit-anonymous-function-parameter-list*opt* )

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-list*opt* )  
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 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

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

* + 1. Операторы

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-listopt }

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

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 ;

* + 1. Пространства имен

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

* + 1. Классы

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-declarationsopt }

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:  
attributesopt constant-modifiersopt 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:  
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

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

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:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

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

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  
;

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:  
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  
;

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  
;

* + 1. Структуры

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

* + 1. Массивы

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

* + 1. Интерфейсы

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 }

* + 1. Перечисления

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:  
attributesopt identifier  
attributesopt identifier = constant-expression

* + 1. Делегаты

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. Атрибуты

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

* 1. Грамматические расширения для небезопасного кода

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:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement   
fixed-statement

unsafe-statement:  
unsafe block

type:  
...  
pointer-type

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

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-listopt

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:  
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 ]

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

1. Ссылки

Консорциум по Юникоду. The Unicode Standard, Version 3.0. Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

IEEE. Стандарт IEEE для двоичной арифметики значений с плавающей запятой. Стандарт ANSI/IEEE 754-1985. Доступен на веб-сайте <http://www.ieee.org>.

ISO/IEC. C++. ANSI/ISO/IEC 14882:1998.