В этой главе рассматриваются три C#-средства, которые позволяют влиять на организацию и доступность программы. Речь пойдет о пространствах имен, препроцессоре и компоновочных файлах.

Пространства имен

О пространствах имен кратко упоминалось в главе 2, поскольку это одно из основополагающих понятий C#: каждая *C#* -программа так или иначе использует некоторое пространство имен. До сих пор мы не затрагивали эту тему, поскольку C# автоматически предоставляет программе пространство имен по умолчанию. Таким образом, программы, приведенные в предыдущих главах, просто использовали стандартное пространство имен. Но реальным программам придется создавать собственные или взаимодействовать с другими пространствами имен. Поэтому настало время поговорить о них более подробно.

Пространство имен определяет декларативную область, которая позволяет отдельно хранить множества имен. По существу, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом. Библиотека .NET Framework (которая является C#-библиотекой) использует пространство имен `System`. Поэтому в начало каждой программы мы включали следующую инструкцию:

```
using System;
```

Как было показано в главе 14, классы ввода-вывода определяются внутри пространства имен, подчиненного `System`, и именуемого `System.IO`. Существуют и другие пространства имен, подчиненные `System`, которые включают иные части C#-библиотеки.

Возникновение пространств имен продиктовано самой жизнью, поскольку в течение последних лет для программирования характерен взрывоподобный рост количества имен переменных, методов, свойств и классов, которые используются в библиотечных процедурах, приложениях сторонних изготовителей ПО и программах, написанных отдельными программистами. Без использования пространств имен все эти имена боролись бы за место “под солнцем” в глобальном пространстве имен, что привело бы к росту числа конфликтов. Например, если в программе определяется класс `Finder`, это имя обязательно будет конфликтовать с именем другого класса, `Finder` из библиотеки стороннего приложения, которое использует ваша программа. К счастью, благодаря пространствам имен, проблем такого рода можно избежать, поскольку пространство имен локализует видимость имен, объявленных внутри него.

Объявление пространства имен

Пространство имен объявляется с помощью ключевого слова `namespace`. Общая форма объявления пространства имен имеет следующий вид:

```
namespace имя {
  // Члены
}
```

Здесь элемент *имя* означает имя пространства имен. Все, что определено внутри пространства имен, находится в пределах его области видимости. Следовательно, пространство имен определяет область видимости. Внутри пространства имен можно объявлять классы, структуры, делегаты, перечисления, интерфейсы или другое пространство имен.

Рассмотрим пример использования ключевого слова namespace, которое создает пространство имен Counter. Оно ограничивает распространение имени, используемого для реализации класса обратного счета, именуемого CountDown.

```
// Объявление пространства имен для счетчиков.
namespace Counter {

  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;
    public CountDown(int n) {
      val = n;
    }
    public void reset(int n) {
      val = n;
    }
    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}
```

Здесь класс CountDown объявляется внутри области видимости, определенной пространством имен Counter.

А теперь рассмотрим программу, которая демонстрирует использование пространства имен Counter.

```
// Демонстрация использования пространства имен.
using System;

// Объявляем пространство имен для счетчиков.
namespace Counter {

  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;
    public CountDown(int n) { val = n; }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    Counter.CountDown cd1 = new Counter.CountDown(10);
    int i;

    do {
```

```
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    Counter.CountDown cd2 = new Counter.CountDown(20);
    do {
      i = cd2. count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
10 9 8 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Здесь имеет смысл обратить ваше внимание вот на что. Во-первых, поскольку класс `CountDown` объявляется внутри пространства имен `Counter`, то при создании объекта класса `CountDown`, как показано в следующей инструкции, имя класса необходимо указывать вместе с именем пространства имен `Counter`.

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

Но если объект `Counter` уже создан, то в дальнейшем называть его (или любой из его членов) полностью (по "имени-отчеству") необязательно. Таким образом, метод `cd1.count()` можно вызывать без указания имени пространства имен, как показано в этой строке кода:

```
i = cd1.count();
```

Пространства имен предотвращают конфликты по совпадению имен

Основное преимущество использования пространств имен состоит в том, что имена, объявленные внутри одного из них, не конфликтуют с такими же именами, объявленными вне его. Например, в следующей программе создается еще один класс `CountDown`, но он находится в пространстве имен `Counter2`.

```
// пространства имен предотвращают конфликты,
// связанные с совпадением имен.
using System;

// Объявляем пространство имен для счетчиков.
namespace Counter {

  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
```

```
    }
    public void reset(int n) {
      val = n;
    }
    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}
// Объявляем еще одно пространство имен.
namespace Counter2 {
  /* Этот класс CountDown находится в пространстве
  имен Counter2 и не будет конфликтовать с одноименным
  классом, определенным в пространстве имен Counter. */

  class CountDown {
    public void count() {
      Console.WriteLine("Этот метод count() находится в" +
             " пространстве имен Counter2.");
    }
  }
}

class NSDemo {
  public static void Main() {

    // Этот класс CountDown находится в
    // пространстве имен Counter.
    Counter.CountDown cd1 = new Counter.CountDown(10);

    // Этот класс CountDown находится в
    // пространстве имен Counter2.
    Counter2.CountDown cd2 = new Counter2.CountDown();
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    cd2.count();
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
10 9 8 7 6 5 4 3 2 1 0
Этот метод count() находится в пространстве имен Counter2.
```

Как подтверждают результаты выполнения этой программы, класс `CountDown` внутри пространства имен `Counter` отделен от класса `CountDown`, определенного в пространстве имен `Counter2`, и поэтому имена не конфликтуют. Хотя этот пример очень простой, он позволяет понять, как избежать конфликтов при совпадении имен между своим кодом и кодом, написанным другими, поместив собственные классы в определенное пространство имен.

Ключевое слово `using`

Как разъяснялось в главе 2, если программа включает часто встречающиеся ссылки на определенные члены пространства имен, то необходимость указывать имя этого пространства имен при каждом к ним обращении, очень скоро утомит вас. Эту проблему позволяет решить директива `using`. В примерах этой книги использовалась директива `using`, чтобы сделать текущим C#-пространство имен `System`, поэтому вы уже с ним знакомы. Нетрудно предположить, что директиву `using` можно также использовать для объявления действующими пространств имен, создаваемых программистом.

Существует две формы применения директивы `using`. Первая имеет такой вид:

```
using имя;
```

Здесь элемент *имя* означает имя пространства имен, к которому необходимо получить доступ. С этой формой директивы `using` вы уже знакомы. Все члены, определенные внутри заданного пространства имен, становятся частью этого (текущего) пространства имен, поэтому их можно использовать без дополнительного упоминания его имени. Директива `using` должна находиться в начале каждого программного файла, т.е. предшествовать всем остальным объявлениям.

В следующей программе переработан пример использования счетчиков из предыдущего раздела, чтобы показать, как с помощью директивы `using` можно чтобы сделать текущим создаваемое программистом пространство имен.

```
// Демонстрация использования пространства имен.
using System;

// Делаем текущим пространство имен Counter.
using Counter;

// Объявляем пространство имен для счетчиков.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;
    public CountDown(int n) {
      val = n;
    }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {

    // Теперь класс CountDown можно использовать
    // без указания имени пространства имен.
    CountDown cd1 = new CountDown(10);
```

```
    int i;
    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    cd2.reset(4);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Эта программа иллюстрирует еще один важный момент: использование одного пространства имен не аннулирует другое. При объявлении действующим некоторого пространства имен его имя просто добавляется к именам других, которые действуют в данный момент. Следовательно, в этой программе действуют пространства имен `System` и `Counter`.

Вторая форма использования директивы `using`

Директива `using` обладает еще одной формой применения:

```
using псевдоимя = имя;
```

Здесь элемент *псевдоимя* задает еще одно имя для класса или пространства имен, заданного элементом *имя*. Теперь программу счета в обратном порядке переделаем еще раз, чтобы показать, как создается *псевдоимя* `Count` для составного имени `Counter.CountDown`.

```
// Демонстрация использования псевдоимени.

using System;

// Создаем псевдоимя для класса Counter.CountDown.
using Count = Counter.CountDown;

// Объявляем пространство имен для счетчиков.
namespace Counter {

  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;
    public CountDown(int n) {
      val = n;
    }
    public void reset(int n) {
```

```
      val = n;
    }
    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    // Здесь Count используется в качестве имени
    // вместо Counter.CountDown.
    Count cd1 = new Count(10);
    int i;
    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    Count cd2 = new Count(20);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

После того как имя `Count` было определено в качестве еще одного имени для составного имени `Counter.CountDown`, его можно использовать для объявления объектов класса `CountDown` без уточняющего указания пространства имен. Например, в нашей программе при выполнении строки

```
Count cd1 = new Count(10);
```

создается объект класса CountDown.

Аддитивность пространств имен

В одной программе одно и то же пространство имен можно объявить больше одного раза. Это позволяет распределить его по нескольким файлам или даже разделить его внутри одного файла. Например, в следующей программе определяется два пространства имен `Counter`. Одно содержит класс `CountDown`, второе — класс `CountUp`. При компиляции содержимое двух пространств имен `Counter` объединяется в одно.

```
// Демонстрация аддитивности пространств имен.

using System;
```

```
// Делаем "видимым" пространство имен Counter.
using Counter;

// Теперь действующим является первое пространство
// имен Counter.
namespace Counter {
   // Простой счетчик для счета в обратном направлении.
   class CountDown {
      int val;
      public CountDown(int n) {
         val = n;
      }
      public void reset(int n) {
         val = n;
      }
      public int count() {
         if(val > 0) return val--;
         else return 0;
      }
   }
}

// Теперь действующим является второе пространство
// имен Counter.
namespace Counter {

// Простой счетчик для счета в прямом направлении.
   class CountUp {
      int val;
      int target;
      public int Target {
         get{
            return target;
         }
      }
      public CountUp(int n) {
         target = n;
         val = 0;
      }
      public void reset(int n) {
         target = n;
         val = 0;
      }

      public int count() {
         if(val < target) return val++;
         else return target;
      }
   }
}

class NSDemo {
   public static void Main() {
```

```
    CountDown cd = new CountDown(10);
    CountUp cu = new CountUp(8);
    int i;
    do {
      i = cd.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
    do {
      i = cu.count();
      Console.Write(i + " ");
    } while(i < cu.Target);
  }
}
```

При выполнении этой программы получаем следующие результаты:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Хотелось бы обратить ваше внимание вот на что. Инструкция

```
using Counter;
```

делает "видимым" содержимое обоих пространств имен. Поэтому к методам `CountDown` и `CountUp` можно обращаться напрямую, без уточняющей информации о пространстве имен. И тот факт, что пространство имен `Counter` было разделено на две части, не имеет никакого значения.

Пространства имен могут быть вложенными

Одно пространство имен можно вложить в другое. Рассмотрим следующую программу:

```
// Демонстрация вложенных пространств имен.
using System;

namespace NS1 {
  class ClassA {

    public ClassA() {
      Console.WriteLine("Создание класса ClassA.");
    }
  }

  namespace NS2 { // Вложенное пространство имен.
    class ClassB {

      public ClassB() {
        Console. WriteLine("Создание класса ClassB.");
      }
    }
  }
}

class NestedNSDemo {
  public static void Main() {
    NS1.ClassA a = new NS1.ClassA();
    // NS2.ClassB b = new NS2.ClassB(); // Ошибка!!!
```

```
      // Пространство имен NS2 не находится в зоне видимости.

      NS1.NS2.ClassB b = new NS1.NS2.ClassB();   // Здесь все
                                                 // правильно.
   }
}
```

Вот результаты выполнения этой программы:

```
Создание класса ClassA.
Создание класса ClassB.
```

В этой программе пространство имен `NS2` вложено в пространство имен `NS1`. Следовательно, при обращении к классу `ClassB` его имя необходимо уточнять, указывая оба пространства имен: как `NS1`, так и `NS2`. Одного лишь имени `NS2` недостаточно. Как видно в программе, имена пространств имен разделяются точкой,

Вложенные пространства имен можно задавать с помощью одной инструкции, но разделив их точками. Например, задание вложенных пространств имен

```
namespace OuterNS {
   namespace InnerNS {
      // ...
   }
}
```

можно записать в таком виде:

```
namespace OuterNS.InnerNS {
   // ....
}
```

Пространство имен по умолчанию

Если для программы не объявлено пространство имен, используется пространство имен, действующее по умолчанию. Вот почему необязательно было указывать его для программ, приведенных в предыдущих главах. Но если для коротких простых программ (подобных тем, что приведены в этой книге) такой стандартный подход вполне приемлем (и даже удобен), большинство реальных программ содержится внутри некоторого пространства имен. Главная причина инкапсуляции программного кода внутри пространства имен состоит в предотвращении конфликтов при совпадении имен. Пространства имен — это еще один инструмент, позволяющий программисту так организовать свои программы, чтобы они не теряли жизнеспособности в сложной среде с сетевой структурой.

Препроцессор

В C# определен ряд директив препроцессора, которые влияют на способ интерпретации исходного кода компилятором. Эти директивы обрабатывают текст исходного файла, в котором они находятся, еще до трансляции программы в объектный код. Директивы препроцессора — в основном “выходцы” из C++, поскольку препроцессор C# очень похож на тот, который определен в языке C++. Термин *директива препроцессора* своим происхождением обязан тому факту, что эти инструкции традиционно обрабатывались на отдельном этапе компиляции, именуемой *процессором предварительной обработки,* или *препроцессором* (preprocessor). Современная технология компиляторов больше не требует отдельного этапа для обработки директив препроцессором, но название осталось.

В C# определены следующие директивы препроцессора:

```
#define      #elif       #else       #endif
#endregion   #error      #if         #line
#region      #undef      #warning
```

Все директивы препроцессора начинаются со знака "#", Кроме того, каждая директива препроцессора должна занимать отдельную строку.

Откровенно говоря, поскольку в C# использована современная объектно-ориентированная архитектура, в директивах препроцессора программисты не испытывают острой необходимости, как это было в языках программирования более ранних поколений. Тем не менее время от времени они могут быть полезными, особенно для условной компиляции. Рассмотрим все перечисленные выше директивы.

#define

Директива `#define` определяет последовательность символов, именуемую *идентификатором.* С помощью директив `#if` или `#elif` можно определить наличие или отсутствие в программе идентификатора, а результат такой проверки используется для управления компиляцией. Общая форма записи директивы `#define` такова:

```
#define идентификатор
```

Обратите внимание на то, что в инструкции нет завершающей точки с запятой. Между самой директивой `#define` и идентификатором может стоять любое количество пробелов, но завершить идентификатор можно только символом новой строки. Например, чтобы определить идентификатор `EXPERIMENTAL`, используйте следующую директиву:

```
#define EXPERIMENTAL
```

На заметку

В C/C++ директиву `#define` можно использовать для выполнения текстовых подстановок, определяя для заданного значения осмысленное имя, а также для создания макросов, действующих подобно функциям. Таков использование директивы `#define` C# не поддерживает. В C# директива `#define` используется только для определения идентификатора.

#if и #endif

Директивы `#if` и `#endif` позволяют выполнить условную компиляцию последовательности инструкций программного кода в зависимости от того, истинно ли выражение, включающее одно или несколько идентификаторов. Истинным считается идентификатор, определенный в программе. В противном случае он считается ложным. Следовательно, если символ определен с помощью директивы `#define`, он оценивается как истинный.

Общая форма использования директивы `#if` такова:

```
#if символьное_выражение
  последовательность_инструкций
#endif
```

Если выражение, стоящее после директивы `#if(символьное_выражение)`, истинно, код, расположенный между нею и директивой `#endif(последовательность_инструкций)`, компилируется. В противном случае он опускается. Директива `#endif` означает конец `#if`-блока.

Символьное выражение может состоять из одного идентификатора. Более сложное выражение можно образовать с помощью следующих операторов: `!`, `==`, `!=`, `&&` и `||`. Разрешено также использовать круглые скобки.

Рассмотрим пример использования директив `#if`, `#endif` и `#define`.

```
// Демонстрация использования директив #if, #endif
// и #define.

#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
        "Компилируется для экспериментальной версии.");
    #endif

    Console.WriteLine(
      "Эта информация отображается во всех версиях.");
  }
}
```

При выполнении программы отображаются следующие результаты:

```
Компилируется для экспериментальной версии.
Эта информация отображается во всех версиях.
```

В этой программе с помощью директивы `#define` определяется идентификатор `EXPERIMENTAL`. Поэтому при использовании директивы `#if` символьное выражение `EXPERIMENTAL` оценивается как истинное, и компилируется первая (из двух) `WriteLine()`-инструкция. Если удалить определение идентификатора `experimental` и перекомпилировать программу, первая `WriteLine()`-инструкция не скомпилируется, поскольку результат выполнения директивы `#if` будет оценен как ложный. Вторая `WriteLine()`-инструкция скомпилируется обязательно, поскольку она не является частью `#if`-блока. Как упоминалось выше, в директиве `#if` можно использовать символьное выражение. Вот пример:

```
// Использование символьного выражения.
#define EXPERIMENTAL
#define TRIAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
          "Компилируется для экспериментальной версии.");
    #endif

    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine(
          "Тестирование экспериментальной пробной версии.");
    #endif
```

```
    Console.WriteLine(
         "Эта информация отображается во всех версиях.");
  }
}
```

Вот результаты выполнения этой программы:

```
Компилируется для экспериментальной версии.
Тестирование экспериментальной пробной версии.
Эта информация отображается во всех версиях.
```

В этом примере определяются два идентификатора, `EXPERIMENTAL` и `TRIAL`. Вторая `WriteLine()`-инструкция компилируется только в случае, если определены оба идентификатора.

`#else` и `#elif`

Директива `#else` работает подобно else-инструкции в языке C#, т.е. она предлагает альтернативу на случай, если директива `#if` выявит ложный результат. Следующими пример представляет собой расширенный вариант предыдущего.

```
// Демонстрация использования директивы #else.
#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
           "Компилируется для экспериментальной версии.");
    #else
      Console.WriteLine("Компилируется для бета-версии.");
    #endif

    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine(
           "Тестирование экспериментальной пробной версии.");
    #else
      Console.Error.WriteLine(
           "Это не экспериментальная пробная версия.");
    #endif

    Console.WriteLine(
           "Эта информация отображается во всех версиях.");
  }
}
```

При выполнении этой программы получены такие результаты:

```
Компилируется для экспериментальной версии.
Это не экспериментальная пробная версия.
Эта информация отображается во всех версиях.
```

Поскольку идентификатор `TRIAL` не определен, компилируется `#else`-блок второй условной последовательности инструкций.

Обратите внимание на то, что директива `#else` отмечает одновременно как конец `#if`-блока, так и начало `#else`-блока, поскольку с любой директивой `#if` может быть связана только одна директива `#endif`.

Директива `#elif` означает “иначе если” и используется в `if-else-if`-цепочках многовариантной компиляции. С директивой `#elif` связано символьное выражение. Если оно истинно, следующий за ним блок кода *(последовательность_инструкций)* компилируется, и другие `#elif`-выражения не проверяются, В противном случае тестируется следующий `#elif`-блок. Общая форма цепочки `#elif`-блоков имеет следующий вид:

```
#if символьное_выражение
  последовательность_инструкций
#elif символьное_выражение
  последовательность_инструкций
#elif символьное_выражение
  последовательность_инструкций
#elif символьное_выражение
  последовательность_инструкций
#elif символьное_выражение
  ⋮
#endif
```

Рассмотрим пример:

```
// Демонстрация использования директивы #elif.
#define RELEASE
using System;
class Test {
  public static void Main() {
    #if EXPERIMENTAL
      Console.WriteLine(
           "Компилируется для экспериментальной версии.");
    #elif RELEASE
      Console.WriteLine("Компилируется для бета-версии.");
    #else
      Console.WriteLine(
           "Компилируется для внутреннего тестирования.");
    #endif
    #if TRIAL && !RELEASE
      Console.WriteLine("Пробная версия.");
    #endif
      Console.WriteLine(
           "Эта информация отображается во всех версиях.");
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Компилируется для бета-версии.
Эта информация отображается во всех версиях.
```

#undef

Директива `#undef` аннулирует приведенное выше определение идентификатора, который указан после директивы. Общая форма директивы `#undef` имеет следующий вид:

```
#undef идентификатор
```

Рассмотрим пример:

```
#define SMALL
#if SMALL
   // ...
#undef SMALL
// Здесь идентификатор SMALL уже не определен.
```

После выполнения директивы `#undef` идентификатор SMALL больше не считается определенным.

Директива `#undef` используется главным образом для того, чтобы разрешить локализацию идентификатора только в пределах нужных разделов кода.

#error

Директива `#error` вынуждает компилятор прекратить компиляцию. Она используется в целях отладки.

Общая форма директивы `#error` имеет следующий вид:

```
#error сообщение_об_ошибке
```

При использовании директивы `#error` отображается заданное *`сообщение_об_ошибке`*. Например, при обработке компилятором строки

```
#error Это тестовая ошибка!
```

процесс компиляции будет остановлен, а на экране появится сообщение “Это тестовая ошибка!”.

#warning

Директива `#warning` подобна директиве `#error`, но она не извещает об ошибке, а содержит предупреждение. Процесс компиляции при этом не останавливается. Общая форма директивы `#warning` имеет следующий вид:

```
#warning предупреждающее_сообщение
```

#line

Директива `#line` устанавливает номер строки и имя файла, который содержит директиву `#line`. Номер строки и имя файла используются во время компиляции при выводе сообщений об ошибках или предупреждений. Общая форма записи директивы `#line` выглядит так:

```
#line номер "имя_файла"
```

Здесь элемент *`номер`* представляет собой любое положительное целое число, которое станет новым номером строки, а необязательный элемент *`имя_файла`* - любой допустимый идентификатор файла, который станет новым именем файла. Директива `#line` в основном используется при отладке и в специальных приложениях.

Чтобы вернуть нумерацию строк в исходное состояние, используйте ключевое слово

```
default: #line default
```

#region и #endregion

Директивы #region и #endregion позволяют определить область, которую можно будет разворачивать или сворачивать при использовании интегрированной среды разработки (IDE) Visual Studio. Вот общая форма использования этих директив:

```
#region имя_области
   // последовательность_инструкций
#endregion
```

Нетрудно догадаться, что элемент *имя_области* означает имя области.

Компоновочные файлы и модификатор доступа internal

Неотъемлемой частью C#-программирования является компоновочный файл (assembly), который содержит информацию о развертывании программы и ее версии. Компоновочные файлы имеют важное значение для .NET-среды. Согласно документации Microsoft, “компоновочные файлы являются строительными блоками среды .NET Framework.” Компоновочные файлы поддерживают механизм безопасного взаимодействия компонентов, межъязыковой работоспособности и управления версиями. Компоновочный файл также определяет область видимости.

Компоновочный файл состоит из четырех разделов. Первый представляет собой *декларацию* (manifest). Декларация содержит информацию о компоновочном файле. Сюда относятся такие данные, как имя компоновочного файла, номер его версии, информация о соответствии типов и параметры “культурного уровня”. Второй раздел включает *метаданные,* или информацию о типах данных, используемых в программе. В числе прочих достоинств метаданных — обеспечение взаимодействия программ, написанных на различных языках программирования. Третий раздел компоновочного файла содержит программный код, который хранится в формате Microsoft Intermediate Language (MSIL). Наконец, четвертый раздел представляет собой ресурсы, используемые программой.

К счастью, при использовании языка C# компоновочные файлы создаются автоматически, без дополнительных (или с минимальными) усилий со стороны программиста. Дело в том, что выполняемый файл, создаваемый в результате компиляции C#-программы, в действительности является компоновочным файлом, который содержит выполняемый код программы и другую информацию. Следовательно, при компиляции C#-программы автоматически создается компоновочный файл.

Подробное рассмотрение компоновочных файлов выходит за рамки этой книги. (Компоновочные файлы — неотъемлемая часть .NET-разработки, а не средство языка C#.) Но одна часть языка C# напрямую связана с компоновочным файлом: модификатор доступа internal. Вот о нем-то и пойдет речь в следующем разделе.

Модификатор доступа internal

Помимо модификаторов доступа public, private и protected, с которыми вы уже встречались в этой книге, в C# также определен модификатор internal. Его назначение — заявить о том, что некоторый член известен во всех файлах, входящих в состав компоновочного, но неизвестен вне его. Проще говоря, член, отмеченный модификатором internal, известен только программе, но не где-то еще. Модификатор доступа internal чрезвычайно полезен при создании программных компонентов.

Модификатор internal можно применить к классам и членам классов, а также к структурам и членам структур. Модификатор internal можно также применить к объявлениям интерфейсов и перечислений.

Совместно с модификатором internal можно использовать модификатор protected. В результате будет установлен уровень доступа protected internal, который можно применять только к членам класса. К члену, объявленному с использованием пары модификаторов protected internal, можно получить доступ внутри его компоновочного файла. Он также доступен для производных типов.

Рассмотрим пример использования модификатора доступа internal.

```
// Использование модификатора доступа internal.

using System;

class InternalTest {
  internal int x;
}

class InternalDemo {

  public static void Main() {
    InternalTest ob = new InternalTest();
    ob.x = 10; // Доступ возможен: x — в том же файле.

    Console.WriteLine("Значение ob.x: " + ob.x);
  }
}
```

Внутри класса InternalTest поле x объявлено с использованием модификатора доступа internal. Это означает, что оно доступно в программе, как показано в коде класса InternalDemo, но недоступно вне ее.