Лекция 6. Наследование

Модификаторы доступа

Модификаторы доступа можно использовать почти с любой определённой сущностью в программе на C#, включая классы и любые члены внутри этих классов. Модификаторы доступа, применяемые к классу, касаются его видимости извне содержащей этот класс сборки. Модификаторы доступа, применённые к членам класса, включая методы, поля, свойства, события и индексаторы, влияют на видимость члена извне данного класса.

В языке C# определены следующие модификаторы доступа:

- `public` — член полностью видим как извне области определения, так и внутри этой области. Другими словами, доступ к общедоступному члену вообще не ограничен;
- `protected` — член видим только определяющему его классу и любому классу-наследнику данного класса;
- `internal` — член видим везде в пределах содержащей его сборки. Сюда входит определяющий его класс и любая область внутри сборки, но вне данного класса;
- `protected internal` — член видим внутри определяющего его класса и везде внутри сборки. Этот модификатор является комбинацией модификаторов `protected` и `internal` с использованием логической операции ИЛИ. Член также видим любому классу-наследнику определяющего его класса, независимо от того, находится он в той же сборке или нет;
- `private` — Член видим только в определяющем классе, без исключений. Это — наиболее строгая форма доступа, и она принята по умолчанию для членов класса.

Члены класса могут использовать все представленные варианты модификаторов доступа C#. Доступом по умолчанию к членам класса в отсутствие всяких модификаторов является `private`. Типы, определенные внутри или вне пространства имён, могут иметь только два модификатора доступа: `public` или `internal`. По умолчанию для них принято `internal`.

Модификаторы доступа `public`, `private` и `internal` можно применять к определению членов `struct`. Модификаторы `protected` и `protected internal` здесь не нужны, поскольку `struct` не поддерживает наследование. Однако, перечисления enum всегда являются общедоступными, поскольку не имеют внутренней реализации, так что ограничение доступа для них не имеет смысла.

Почти всегда доступом по умолчанию является наиболее ограниченный доступ, имеющий смысл для данной сущности. Другими словами, чтобы открыть другой доступ к классам или членам класса, потребуется приложить некоторые усилия. Единственное исключение — доступ к пространству имён, принятый по умолчанию как `public` и не допускающий указания каких-либо модификаторов доступа.

Наследование классов

Создать новый класс можно на основе расширения функциональности одного из существующих классов. Синтаксис такой же, как и при создании нового класса, но после имени класса следует двоеточие и имя базового класса. В C# класс имеет только один базовый класс.

Важным аспектом наследования является доступность членов, особенно, когда речь идёт о доступности членов базового класса из производного класса. Любые общедоступные члены базового класса становятся общедоступными и в производном классе.

Любые члены, помеченные модификатором `protected` (защищённые), доступны только внутри объявляющего их класса и его наследников. Защищённые члены никогда не доступны извне определяющего их класса или его наследников. Приватные (`private`) члены недоступны нигде, кроме определяющего их класса. Поэтому, несмотря на то, что производный класс наследует все члены базового класса, включая и приватные, код в производном классе не имеет доступа к приватным членам, унаследованным от базового класса. Вдобавок защищённые внутренние (`protected internal`) члены также видимы всем типам, определённым внутри одной сборки, и классам-наследникам определившего их класса. Реальность состоит в том, что производный класс наследует все члены базового класса за исключением конструкторов экземпляра, статических конструкторов и деструкторов.

Управлять доступностью всего класса в целом можно при его определении. Единственными вариантами доступа к типу класса являются `internal` и `public`. При использовании наследования действует правило, что тип базового класса должен быть доступен как минимум настолько же, как и производный класс. Рассмотрим следующий код:

```
class A
{
    protected int x;
}
public class B : A
{
}
```

Этот код не скомпилируется, потому что класс `A` объявлен как `internal` и не является настолько (как минимум) доступным, как производный от него класс `B`. При отсутствии модификатора доступа класс имеет доступ `internal`, поэтому класс `A` на самом деле является `internal`. Для того чтобы этот код компилировался, понадобится либо повысить класс `A` до уровня доступа `public`, либо ограничить класс `B` доступом `internal`. К тому же обратите внимание, что для класса `A` допустимо быть `public`, а для класса `B` — `internal`.

Представлять наследование и то, что оно делает, можно несколькими способами. Первый и наиболее очевидный — наследование позволяет позаимствовать реализацию. Другими словами, можно унаследовать класс `D` от класса `A` и повторно использовать реализацию класса `A` в классе `D`. Потенциально это позволит сэкономить некоторую часть работы при определении класса `D`. Другое применение наследования — специализация, когда класс `D` становится специализированной формой класса `A`.

Например, рассмотрим иерархию классов, показанную на рисунке 6.1.

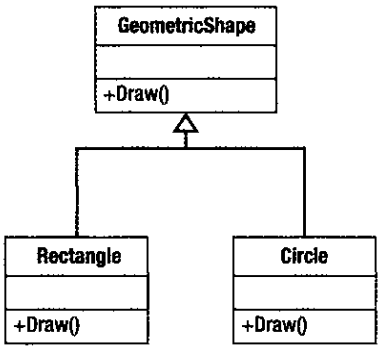

Рис. 6.1. Специализация при наследовании

Классы `Rectangle` и `Circle` наследуются от класса `GeometricShape`. Другими словами, они являются специализациями класса `GeometricShape`. Класс `GeometricShape` имеет свойства, определяющие координаты фигуры, и эти свойства также есть у классов-наследников `Rectangle` и `Circle`. Классы `Rectangle` и `Circle` содержат дополнительные свойства, расширяющие базовый класс: для прямоугольника — ширина и высота, для круга — радиус.

У всех трёх классов есть метод `Draw`, выполняющий отрисовку изображения фигуры, но реализован он должен быть в каждом классе по-своему. В следующем примере показана реализация классов `GeometricShape`, `Rectangle` и `Circle`:

```
public class GeometricShape // класс, представляющий геометрическую фигуру
{
    public int X { get; set; } // координаты
    public int Y { get; set; } // фигуры

    public void Draw()
    {
        // Выполнить отрисовку по умолчанию, например,
        // нарисовать точку с координатами X и Y
    }
}
```

```
public class Rectangle : GeometricShape // класс, представляющий прямоугольник
{
    public int Width { get; set; }  // ширина прямоугольника
    public int Height { get; set; } // высота прямоугольника

    public void Draw()
    {
        // Нарисовать прямоугольник
    }
}

public class Circle : GeometricShape // класс, представляющий круг
{
    public int R { get; set; } // радиус круга

    public void Draw()
    {
        // Нарисовать круг
    }
}

...

GeometricShape point = new GeometricShape()
{
    X = 10,
    Y = 40
};
point.Draw();

Rectangle rect = new Rectangle()
{
    X = 50,
    Y = 20,
    Width = 200,
    Height = 100
};
rect.Draw();

Circle circle = new Circle()
{
    X = 100,
    Y = 100,
    R = 50
};
circle.Draw();
```

В результате выполнения заданного кода будут выведены изображения точки, прямоугольника и круга в соответствии с заданными значениями свойств.

Экземпляры классов-наследников можно присваивать переменным, имеющим тип класса родителя:

```
GeometricShape figure = circle;
circle.X = 150;
circle.Y = 100;
```

После выполнения этой строки кода переменная `figure` будет содержать ссылку на тот же круг, что и в переменной `circle`, и при изменении значений координат объекта `figure` будут изменяться координаты круга `circle`. Однако свойство `R` недоступно в объекте `figure`, поскольку является специфическим для класса-наследника `Circle`.

Сокрытие членов

Несмотря на свою мощность, наследование может быть использовано некорректно. Когда программисты впервые узнают о наследовании, они склонны применять его слишком часто, создавая проекты и иерархические структуры, которые впоследствии трудно сопровождать. Важно отметить, что у наследования есть альтернативы, которые во многих случаях более оправданы. Среди различного рода ассоциаций, возможных между классами в дизайне программной системы, наследование — самая жёсткая из всех. Рассмотрим некоторые основные эффекты от наследования.

Обратите внимание, что наследование расширяет функциональность, но не может её исключать. Например, общедоступные методы базового класса доступны через экземпляры производного класса и классов, унаследованных от него. Удалить эти функциональные возможности из производного класса нельзя. Рассмотрим следующий код:

```
public class A
{
    public void DoSomething()
    {
        Console.WriteLine("A.DoSomething");
    }
}

public class B : A
{
    public void DoSomethingElse()
    {
        Console.WriteLine("B.DoSomethingElse");
    }
}

...

B b = new B();
b.DoSomething();
b.DoSomethingElse();
```

Здесь создаётся новый экземпляр класса `B`, унаследованного от класса `A`. Класс `B` получает объединение членов обоих классов — `A` и `B`. Вот почему можно вызывать и `DoSomething`, и `DoSomethingElse` на экземпляре класса `B`. Это вполне очевидно, поскольку наследование расширяет функциональность.

Но что, если необходимо наследовать от класса `A`, но скрыть метод `DoSomething`? Другими словами, что если требуется расширить лишь часть функциональности `A`?

С помощью наследования это невозможно. Однако есть возможность *сокрытия* члена, как показано в следующем коде, который является модифицированной формой предыдущего примера:

```
public class A
{
    public void DoSomething()
    {
        Console.WriteLine("A.DoSomething");
    }
}

public class B : A
{
    public void DoSomethingElse()
    {
        Console.WriteLine("B.DoSomethingElse");
    }
    public new void DoSomething()
    {
        Console.WriteLine("B.DoSomething");
```

```
    }
}

...

B b = new B();
b.DoSomething();
b.DoSomethingElse();
A a = b;
a.DoSomething();
```

В этой версии в класс `B` введён новый метод по имени `DoSomething`. Также обратите внимание на добавление ключевого слова `new` к объявлению `B.DoSomething`. Если не добавить это ключевое слово, то компилятор выдаст предупреждение. Это его способ сообщить о необходимости выражаться яснее относительно сокрытия метода базового класса. Возможно, компилятор делает так потому, что подобное сокрытие членов обычно трактуется как плохой дизайн. Давайте разберемся, почему. Вывод предыдущего кода выглядит следующим образом:

```
B.DoSomething
B.DoSomethingElse
A.DoSomething
```

Первое, на что следует обратить внимание: то, какой именно метод `DoSomething` будет вызван, зависит от типа ссылки, через которую он будет вызван. Это достаточно неочевидно, поскольку `B` является `A`, а наследование моделирует отношение «является». В данном случае должен ли весь общедоступный интерфейс `A` быть доступным потребителям экземпляра класса `B`? Если кратко, то нет. Когда действительно необходимо, чтобы метод вёл себя по-разному в подклассах, тогда в точке определения класса `A` метод `DoSomething` должен быть объявлен виртуальным. При таком подходе правильнее было бы воспользоваться полиморфизмом. В этом случае должна вызываться самая последняя версия (версия наследника) метода `DoSomething`, независимо от того, через какой тип ссылки он был вызван.

Несмотря на то что класс `B` теперь скрывает реализацию `DoSomething` класса `A`, помните, что он не удаляет её. Он скрывает её при вызове этого метода через ссылку типа `B` на объект. Однако это легко обойти, применив неявное преобразование ссылки на экземпляр типа `B` в ссылку на экземпляр типа `A` с последующим вызовом через неё реализации `A.DoSomething`. То есть реализация `A.DoSomething` не исчезла, она просто скрыта. И чтобы добраться до неё, понадобится просто проделать немного больше работы.

Это — классический пример того, что если язык позволяет делать нечто подобное, это не означает, что так следует поступать. Почти любые существующие языки обладают такими средствами, которые, будучи использованными (к тому же неправильно), лишь привнесут излишнюю сложность и ухудшат дизайн.

Герметизированные классы

Наследование — это мощный инструмент, которым легко злоупотребить. Иногда имеются намерения сделать создаваемый новый класс базовым классом — заготовкой для дальнейшей специализации. Однако часто бывает и так, что классы проектируются, не принимая во внимание то, будут они использоваться в качестве базовых или нет. На самом деле весьма вероятно, что класс, который проектируется сегодня, будет использован в качестве базового завтра, даже если это не предполагалось изначально.

В C# предлагается ключевое слово `sealed` для тех случаев, когда нужно сделать так, чтобы клиент не мог наследовать свой класс от конкретного класса. Применённое к целому классу, ключевое слово `sealed` указывает на то, что данный класс является *листовым* в дереве наследования. Под этим понимается запрет наследования от данного класса. Поначалу может показаться, что ключевое слово `sealed` придётся использовать редко, однако на самом деле должно быть наоборот. При проектировании новых классов это ключевое слово должно применяться настолько часто, насколько возможно. Многие советуют использовать его по умолчанию.

Наследование — такая хитрая штука, что для того, чтобы класс был хорошим базовым классом, он должен сразу проектироваться с таким прицелом. В противном случае сразу помечайте его как `sealed`. Это очень про-

сто. Может также возникнуть вопрос: почему бы не обеспечить максимальную гибкость, оставив класс негерметизированным, чтобы кто-нибудь в будущем мог выполнять наследование от него? При хорошем дизайне это не допускается. Следует еще раз подчеркнуть: класс, предназначенный для того, чтобы служить базовым классом, должен быть спроектирован с учётом этого с самого начала. Если это не так, то весьма вероятно, что наследование от него производного класса будет сопряжено со многими сложностями.

Абстрактные классы

Абстрактные классы диаметрально противоположны классам `sealed`. Иногда необходимо спроектировать класс, единственное назначение которого — служить базовым классом. Подобного рода классы помечаются ключевым словом `abstract`.

Ключевое слово `abstract` сообщает компилятору, что назначение данного класса — служить базовым, и потому создавать экземпляры этого класса не разрешено. Вернёмся к приведённому ранее примеру с геометрическими фигурами. Приведённый ранее код можно изменить следующим образом:

```
public abstract class GeometricShape
{
    public int X { get; set; }
    public int Y { get; set; }

    public abstract void Draw();
}

public class Circle : GeometricShape
{
    public int R { get; set; }

    public override void Draw()
    {
        // Нарисовать круг
    }
}

...

Circle circle = new Circle()
{
    X = 100,
    Y = 100,
    R = 50
};
circle.Draw();

// Этот код не будет работать!
// GeometricShape point = new GeometricShape();
```

Создавать объект `GeometricShape`, вообще говоря, не имеет смысла, поэтому класс `GeometricShape` сделан абстрактным. Таким образом, если в коде попытаться создать экземпляр `GeometricShape`, возникнет ошибка компиляции. Ключевое слово `abstract` также встречается в методе `GeometricShape.Draw`. Оно позволяет сообщить компилятору, что производные классы должны переопределить этот метод. Поскольку метод должен быть переопределён в производных классах, не имеет смысла предусматривать реализацию `GeometricShape.Draw`, раз уж всё равно нельзя создать экземпляр `GeometricShape`. Поэтому абстрактные методы не должны иметь реализации.

Как видите, бывают случаи, когда базовый класс используется для определения некоторого шаблона или поведения, перекладывая ответственность за реализацию на наследников. От этого базового шаблона могут наследоваться листовые классы, уточняя подробности реализации.

Вложенные классы

Вложенные классы определяются внутри области определения другого класса. Классы, определённые внутри контекста пространства имён или вне пространства имён, но не внутри контекста другого класса, называются *невложенными*. Вложенные классы обладают некоторыми специальными возможностями, которые удобны, когда нужен вспомогательный класс, работающий внутри содержащего его класса.

Например, контейнерный класс может содержать коллекцию объектов. Предположим, что требуется некоторое средство для выполнения итерации по всем содержащимся объектам, чтобы позволить внешним пользователям, выполняющим итерацию, поддерживать маркер, или некую разновидность курсора, который запоминает свое текущее место во время итерации. Это распространенный подход в проектировании. Избавление пользователей от необходимости хранить прямые ссылки на содержащиеся в коллекции объекты обеспечивает большую гибкость в отношении изменения внутреннего поведения контейнерного класса без разрушения кода, использующего этот контейнерный класс. Вложенные классы по нескольким причинам предоставляют отличное решение такой проблемы.

Вложенные классы имеют доступ ко всем членам, видимым содержащему их классу, даже если эти члены являются приватными. К вложенным классам можно применять те же модификаторы доступа, что и к любым другим членам класса.

Вложенные классы, объявленные как `public`, позволяют создавать экземпляры коду, внешнему по отношению к охватывающему классу. Нотация обращения к вложенному классу подобна уточнению пространств имён. В следующем примере демонстрируется создание экземпляра вложенного класса:

```
public class A
{
    public class B
    {
        ...
    }
}

...

A.B b = new A.B();
```

Иногда имя нового вложенного класса может скрыть имя члена внутри базового класса за счёт использования ключевого слова `new`, подобно тому, как это делается при сокрытии методов. Это бывает очень редко, и в большинстве случаев данной ситуации можно избежать. Рассмотрим следующий пример:

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
{
    public new class Foo
    {
    }
}
```

В этом случае вложенный класс `Foo` определяется внутри определения класса `B`. Поскольку его имя совпадает с именем метода `Foo` в классе `A`, вы обязаны применить ключевое слово `new`, в противном случае компилятор сообщит о конфликте имён. Если подобная ситуация возникает, то это означает, скорее всего, что наступило время переосмыслить дизайн или просто переименовать вложенный класс, если только в намерения не входит действительное сокрытие базового члена. Сокрытие базовых членов подобным образом — сомнительное решение, которого обычно не стоит придерживаться лишь по той причине, что язык позволяет это.

Статические классы

В C# 2.0 появился новый модификатор классов, позволяющий указывать, что данный класс — не что иное, как коллекция статических членов, и создавать его экземпляры не разрешено. Для этого к классу должен быть применен модификатор `static`. В таком случае на класс накладываются следующие ограничения:

- класс не может наследоваться ни от чего кроме `System.Object`, и даже если этот базовый тип не указан, наследование от него подразумевается;
- класс не может использоваться в качестве базового для других классов;
- класс может содержать только статические члены, которые могут быть общедоступными или приватными. Однако они не могут помечаться как `protected` или `protected internal`, поскольку класс не может служить базовым;
- класс не может иметь никаких операций, поскольку, по причине невозможности создания экземпляров класса, их определение не имеет смысла.

Несмотря на то что весь класс помечен как `static`, все его индивидуальные члены также помечаются с помощью `static`. Хотя компилятору было бы просто предположить, что любой член внутри статического класса также является статическим, это внесло бы излишнюю сложность в без того сложный компилятор. Если вы примените модификатор `static` к вложенному классу, он также будет статическим, как и содержащий его класс, а вложенные классы, не снабженные модификатором `static`, допускают создание их экземпляров.

Ниже приведён пример статического класса:

```
public static class StaticClass
{
    private static long callCount = 0;
    public static long CallCount
    {
        get { return callCount; }
    }

    public static void DoWork()
    {
        ++callCount;
        Console.WriteLine("StaticClass.DoWork()");
    }

    public class NestedClass
    {
        public NestedClass()
        {
            Console.WriteLine("NestedClass.NestedClass()");
        }
    }
}

...

StaticClass.DoWork();
// Так поступать нельзя!
// StaticClass obj = new StaticClass();
StaticClass.NestedClass nested = new StaticClass.NestedClass();
Console.WriteLine("CallCount = {0}", StaticClass.CallCount);
```

Тип `StaticClass` содержит метод, поле, свойство и вложенный класс. Обратите внимание, что поскольку `NestedClass` не объявлен как `static`, можно создавать его экземпляры. К тому же, из-за того, что класс `EntryPoint` просто содержит метод `Main`, он также помечен как `static`, чтобы предотвратить непреднамеренное создание его экземпляров.

Статические классы удобны, когда необходим логический механизм организации набора методов. Примером статического класса из библиотеки базовых классов может служить знаменитый класс `System.Console`.

Он поддерживает методы, свойства и события, которые все являются статическими, так как в один и тот же момент времени к процессу может быть присоединена только одна консоль.