Лекция 5. Создание классов

Как уже говорилось ранее, с точки зрения CLR и языка программирования C#, любой тип является классом. Создаваемые через определение классов объекты C# обладают теми же возможностями, что и другие предопределённые в системе объекты. Фактически, ключевые слова языка C# вроде `int` и `bool` — это просто псевдонимы для предопределённых типов значений из пространства имен `System`, в данном случае — `System.Int32` и `System.Boolean` соответственно.

Мощь объектно-ориентированных систем заключается в возможности создания разработчиками собственных типов. Несмотря на то, что встроенные типы языка являются простыми объектами CLR, создаваемые вами объекты обращаются со встроенными типами на уровне полей. Другими словами, встроенные типы не обладают никакими специальными возможностями, которые нельзя было бы реализовать в типах, определяемых пользователем.

Объекты, создаваемые на основе классов, инкапсулируют поля данных, представляющие внутреннее состояние объектов, и объекты могут тонко управлять доступом к этим полям. Поведение объектов определяется реализуемыми методами, которые объявляются и определяются внутри определения класса. Вызывая один из методов на экземпляре объекта, вы инициируете единицу работы этого объекта. Эта работа может модифицировать внутреннее состояние объекта, проверить состояние объекта либо сделать что-то другое в том же духе.

В классе можно определить конструкторы, которые система выполняет при каждом создании нового объекта. Кроме того, можно определить метод, называемый *финализатором*, который будет работать во время удаления объекта сборщиком мусора. Однако, финализаторов следует по возможности избегать.

Объекты поддерживают концепцию *наследования*, посредством которой производный класс наследует поля и методы базового класса. Наследование также позволяет трактовать объекты производного класса как объекты его базового типа. Например, дизайн, в котором объект типа `Dog` унаследован от типа `Animal`, говорит о том, что это модель отношения «является» (т.е. собака (`Dog`) является животным (`Animal`)). Таким образом, можно просто неявно преобразовать ссылку на тип `Dog` в ссылку на тип `Animal`. Здесь неявно означает, что преобразование имеет форму простого выражения присваивания. В противоположность этому, ссылку на тип `Animal` можно преобразовать явно через операцию приведения, чтобы она ссылалась на тип `Dog`, если определённый объект, на который ссылается тип `Animal`, фактически является объектом, созданным на основе класса `Dog`. Эта концепция называется *полиморфизмом* — когда можно манипулировать объектами взаимосвязанных типов, как если бы они относились к одному общему типу.

Среда CLR отслеживает ссылки на объекты. Это значит, что каждая переменная ссылочного типа в действительности содержит ссылку на объект в куче (или `null`, если в данный момент не ссылается на какой-либо объект). В результате копирования значения переменной ссылочного типа в другую переменную ссылочного типа создается другая ссылка на тот же самый объект — другими словами, копируется ссылка. Таким образом, получаются две переменных, ссылающихся на один и тот же объект. В CLR для создания полноценных копий объектов понадобится проделать дополнительную работу, например, реализовать интерфейс `ICloneable` или аналогичный шаблон.

Все объекты, созданные на основе определения класса C#, располагаются в системной куче, которой управляет сборщик мусора (Garbage Collector — GC) среды CLR. Сборщик мусора избавляет от необходимости очистки памяти, занятой объектами. GC достаточно интеллектуален, чтобы отслеживать ссылки на объекты, и когда он обнаруживает, что на объект не осталось никаких ссылок, он помечает его для удаления. Затем, когда в следующий раз GC уплотняет кучу, он уничтожает такой объект и освобождает занятую им память.

Наряду с классами, язык C# также поддерживает определение новых *типов значений* через ключевое слово `struct`. Типы значений — это легковесные объекты, которые обычно не находятся в куче, а вместо этого располагаются в стеке. Вообще, тип значений может находиться в куче, но только в том случае, если он является полем внутри объекта, расположенного в куче. Типы значений не могут быть определены так, чтобы наследоваться от другого класса или другого типа значений, равно как и никакой другой класс или тип значений не может наследоваться от них.

Типы значений могут иметь конструкторы, но не финализаторы. По умолчанию при передаче методу типа значения в качестве параметра метод принимает копию этого значения.

Определения классов в C# похожи на определения классов в C++. Новый класс объявляется с помощью ключевого слова `class`:

```
public class MyClass { }
```

Модификатор доступа перед ключевым словом `class` управляет видимостью типа извне сборки. Класс `MyClass` является общедоступным, а это означает, что пользователи сборки, содержащей этот класс, могут создавать его экземпляры.

Поля

Поля (*field*) — это механизм, представляющий состояние объектов. Обычно новый класс объявляется только тогда, когда нужно смоделировать новый тип объекта, с собственным внутренним состоянием, представленным его полями экземпляра.

Инициализировать поля во время создания объекта можно различными способами. Простейший способ сделать это — прибегнуть к помощи *инициализаторов*. Они задаются в точке определения поля и могут применяться как для статических полей, так и для полей экземпляра, например:

```
int x = 78 9;
int y;
int z = A.InitZ();
```

Поле x инициализируется инициализатором. Эта инициализация происходит во время выполнения, а не во время компиляции. Поэтому в данном операторе инициализации могут использоваться не только константы. Например, переменная z инициализируется вызовом метода `A.InitZ`. Поначалу такое обозначение инициализации поля может показаться значительным сокращением, избавляющим от необходимости инициализировать все поля внутри тела конструктора. Однако, для объявлений сложных типов всё-таки рекомендуется инициализировать поля экземпляра внутри тела конструктора экземпляра, поскольку инициализация полей в конструкторе облегчает создание кода, который проще сопровождать и отлаживать.

Часто при описании полей используется модификатор `readonly`. Как и можно было ожидать, он позволяет определить поле, которое доступно только для чтения. Осуществлять запись в такое поле можно только при создании объекта. В качестве альтернативы такие поля можно инициализировать посредством инициализаторов в точке их объявления в классе, как это делается с другими полями.

В C# поддерживаются определённые правила инициализации полей по умолчанию, которые применяются перед выполнением кода инициализации, встречающегося в блоке кода методов конструкторов. По умолчанию C# создаёт верифицируемый безопасный к типам код, который гарантированно не использует неинициализированные переменные и поля. Компилятор тщательно следит за удовлетворением этого требования. Так, например, он инициализирует все поля, будь то поля экземпляра или статические, значениями по умолчанию, причём это делается до запуска любых инициализаторов. Значение по умолчанию почти для чего угодно может быть представлено либо как 0, либо как `null`. Такая инициализация по умолчанию происходит перед выполнением любого кода на экземпляре класса. Таким образом, опрос неинициализированных значений объекта или класса во время начального конструирования невозможен.

Конструкторы

Конструкторы вызываются при первоначальной загрузке класса средой CLR или при создании объекта. Синтаксически конструктор представляет собой метод, имя которого совпадает с именем класса. Существуют два типа конструкторов: *статические конструкторы* и *конструкторы экземпляра*. Класс может иметь только один статический конструктор, не имеющий параметров. Он вызывается, когда CLR загружает тип.

С другой стороны, конструкторы экземпляра вызываются при создании экземпляра класса. Обычно они устанавливают состояние объекта за счёт инициализации полей в желательное предопределённое состояние. Можно также предпринять любые другие действия по инициализации, такие как подключение к базе данных и открытие файла. У класса может быть несколько конструкторов экземпляра, которые могут быть *перегружены* (т.е. иметь разные типы параметров). Подобно статическим конструкторам, конструкторы экземпляра именуются по названию определяющего их класса:

```
public class MyClass
{
    public int x = 15;

    public MyClass(int x)
    {
        this.x = x;
    }
}
```

В этом примере имя параметра конструктора класса `MyClass` совпадает с именем поля этого класса. Для устранения неоднозначности между параметром и переменной используется ключевое слово `this`. Оно представляет собой ссылку на текущий экземпляр класса, т.е. объект, который создаётся с помощью этого конструктора. В общем случае, ключевое слово `this` можно использовать в любом методе, который вызывается для экземпляра класса.

Для использования класса необходимо создать его экземпляр и присвоить ссылку на него переменной соответствующего типа.

```
MyClass obj = new MyClass(10);
Console.WriteLine(obj.x); // x = 10
```

В этом примере ключевое слово `new` вызывает конструктор класса `MyClass`, которому передаётся параметр типа `int`, после чего выводится на экран значение поля x — в данном случае значение 10, поскольку код внутри конструктора выполняется после инициализации поля x значением 15.

Если в описании класса не определён конструктор, то автоматически создаётся конструктор без параметров:

```
public class MyClass
{
    public int x = 15;
}

...

MyClass obj = new MyClass();
Console.WriteLine(obj.x); // x = 15
```

Методы

Метод определяет процедуру, которую можно выполнить над объектом или классом. Если метод является методом *экземпляра*, то его можно вызывать на объекте. Если же метод *статический*, его можно вызывать только на классе. Отличие между ними в том, что метод экземпляра имеет доступ и к полям экземпляра объекта, и к статическим полям класса, в то время как статический метод не имеет доступа к полям и методам экземпляра. Статические методы могут иметь доступ только к статическим членам класса.

Методы могут иметь присоединённые к ним атрибуты метаданных, а также необязательные модификаторы, которые управляют доступностью методов, а также улучшают методы, подходящие для наследования. Каждый метод либо имеет, либо не имеет типа возврата. Если метод не имеет типа возврата, то в его объявлении в качестве такого типа должно указываться `void`. Методы могут иметь параметры.

Статические методы вызываются на всём классе, а не на его экземплярах. Статические методы имеют доступ только к статическим членам класса. Такие методы объявляются с применением модификатора `static`:

```
public class MyClass
{
    public static void SomeFunction()
    {
        Console.WriteLine("Вызван метод SomeFunction()");
    }
```

```
    static void Main()
    {
        MyClass.SomeFunction();
        SomeFunction();
    }
}
```

Обратите внимание, что оба метода в этом примере являются статическими. В методе `Main` сначала вызывается метод `SomeFunction` с указанием имени класса. После этого вызывается статический метод без указания имени класса. Причина в том, что методы `Main` и `SomeFunction` определены в одном и том же классе и оба являются статическими. Если бы метод `SomeFunction` относился к другому классу, скажем, `MyNewClass`, на этот метод пришлось бы ссылаться следующим образом: `MyNewClass.SomeFunction`.

Методы экземпляра работают с объектами. Для того чтобы вызвать метод экземпляра, необходимо сослаться на экземпляр класса, определяющего этот метод:

```
    public class MyClass
    {
        int x;
        int y;
        static int z;

        void SomeOperation()
        {
            x = 1;
            this.y = 2;
            z = 3;
        }

        static void Main()
        {
            MyClass obj = new MyClass();
            obj.SomeOperation();
            Console.WriteLine("x = {0}, y= {1}, z= {2}", obj.x, obj.y, MyClass.z);
        }
    }
```

В методе `Main` создается новый экземпляр класса `MyClass` и затем вызывается метод `SomeOperation` через экземпляр этого класса. Внутри тела метода `SomeOperation` имеется доступ к полям экземпляра и статическим полям класса, поэтому присваивать им значения можно просто с использованием их идентификаторов. Несмотря на то, что в методе `SomeOperation` можно присваивать значение статическому члену `z`, не квалифицируя его, для достижения лучшей читабельности кода во время присваивания значений следует квалифицировать статические поля даже в методах, относящихся к тому же самому классу.

Обратите внимание, что при присваивании значения поле `y` снабжается префиксом — идентификатором `this`. С помощью `this` можно обращаться к полям экземпляра, как было показано ранее. Поскольку значение `this` доступно только для чтения, ему нельзя присвоить что-либо, что заставит его ссылаться на другой экземпляр. При попытке сделать это компилятор сообщит об ошибке, и код не скомпилируется.

Свойства

Свойства используются для ужесточения контроля доступа к внутреннему состоянию объекта. С точки зрения клиента объекта свойство выглядит и ведёт себя аналогично общедоступному полю. Синтаксис доступа к свойству такой же, как при доступе к общедоступному полю экземпляра. Однако свойство не имеет никакого ассоциированного с ним места хранения в объекте, как это присуще полям. Вместо этого свойство представляет собой сокращённый синтаксис для определения *средств доступа* (*accessors*) для чтения и записи полей. Типичный шаблон предусматривает обеспечение доступа к закрытому полю класса через общедоступное свойство. Начиная с версии языка C# 3.0 эта задача ещё более облегчается за счёт введения *автореализуемых* (*auto-implemented*) свойств.

Свойства существенно расширяют возможности проектировщиков классов. Например, если свойство представляет количество строк в объекте, который связывается с базой данных, то этот объект может отложить вычисление этого значения до тех пор, пока оно не будет опрошено свойством. Объект таблицы базы данных узнаёт, когда необходимо вычислить значение, по факту вызова клиентом средства доступа для обращения к свойству.

В следующем коде определяется свойство X в классе MyClass, после чего создаётся и используется экземпляр этого класса:

```
public class MyClass
{
    private int x;
    public int X
    {
        get
        {
            Console.WriteLine("Извлечение значения x");
            return x;
        }
        set
        {
            Console.WriteLine("Установка значения x");
            x = value;
        }
    }
}

...

MyClass obj = new MyClass();
obj.X = 1;
Console.WriteLine("obj.X = {0}", obj.X);
```

Здесь сначала определяется свойство по имени X, имеющее тип int. Каждое объявление свойства должно определять тип, представляемый этим свойством. Этот тип должен быть видимым компилятору в точке, где свойство объявлено в классе, и он должен иметь, как минимум, ту же видимость, что и определяемое свойство. То есть имеется в виду, что если свойство является общедоступным (public), то тип значения, представленного свойством, также должен быть объявлен как public в сборке, в которой он определён. В данном примере тип int — это псевдоним для System.Int32. Этот класс определён в пространстве имен System, и он объявлен как public. Поэтому int можно использовать в качестве типа свойства в данном общедоступном классе MyClass.

Свойство X просто возвращает приватное поле x из внутреннего состояния экземпляра объекта. Это общепринятое соглашение. Приватному полю назначается имя, начинающееся с прописной буквы, а свойству — имя, начинающееся с заглавной буквы. Разумеется, следовать этому соглашению не обязательно, но нет причин от него отказываться, к тому же другие программисты C# обычно ожидают его соблюдения.

Средства доступа

В предыдущем примере внутри блока свойства находятся ещё два блока кода — это *средства доступа* (*accessors*) к свойству, и внутри их блоков помещается код, который читает и записывает значение свойства. Соответственно, первый блок называется get, а другой — set.

Блок get вызывается, когда клиент объекта читает свойство. Как и можно было ожидать, это средство доступа должно возвращать значение или ссылку на объект, соответствующий типу объявления свойства. Также он возвращает объект, неявно преобразуемый в объявленный тип свойства. Например, если типом свойства является long, а get возвращает int, то тип int будет неявно преобразован в long без потери точности. В других отношениях код в этом блоке подобен параметризованному методу, возвращающему значение или ссылку на тип свойства.

Блок `set` вызывается, когда клиент пытается записывать свойство. Обратите внимание, что возвращаемого значения здесь нет. Также обратите внимание на специальное значение по имени `value`, которое доступно коду внутри этого блока, и которое имеет тот же тип, что объявлен в качестве типа свойства. При записи значения в свойство переменная `value` устанавливается в значение или ссылку на объект, которую клиент пытается присвоить свойству. Если вы попробуете объявить локальную переменную по имени `value` в блоке `set`, то получите ошибку компиляции. Средство доступа `set` подобно методу, который принимает один параметр того же типа, что у свойства, и возвращает `void`.

Если свойство определяется только со средством доступа `get`, то такое свойство будет доступно только для чтения. Аналогично если свойство определяется только со средством доступа `set`, то получается свойство, доступное только для записи. И, наконец, свойство, имеющее оба средства доступа, допускает как чтение, так и запись.

Может возникнуть вопрос: чем свойство, доступное только для чтения, лучше обычного общедоступного поля с модификатором `readonly`? На первый взгляд может показаться, что доступное только для чтения свойство менее эффективно доступного только для чтения общедоступного поля. Однако, учитывая тот факт, что во время JIT-компиляции CLR может встраивать код для доступа к свойству в случае, когда свойство просто возвращает поле, проблема неэффективности отпадает. После этого может показаться, что запись кода не будет особенно эффективной. Однако поскольку программисты не так уж ленивы, а автореализуемые свойства существенно облегчают задачу записи, данный аргумент также не слишком убедителен.

Фактически в 99 % случаев свойство, доступное только для записи, является более гибким, чем общедоступное `readonly`-поле. Одна из причин связана с возможностью отложить вычисление свойства, доступного только для чтения, до того момента, когда оно понадобится (прием, известный как *«ленивое» вычисление* или *отложенное выполнение*). В действительности это может привести к более эффективному коду, когда свойство предназначено для представления чего-то такого, на вычисление чего требуется существенное время. Если для этой цели используется общедоступное `readonly`-поле, то придётся выполнить вычисление в блоке конструктора. Все необходимые для вычисления свойства к этому моменту могут быть еще не готовы. Или же можно потратить время на вычисление значения в конструкторе, хотя пользователь объекта, возможно, никогда не обратится к этому значению.

Вдобавок свойства, доступные только для чтения, помогают усилить инкапсуляцию. Если изначально имеется выбор между доступным только для чтения свойством и общедоступным `readonly`-полем, то, отдав предпочтение свойству только для чтения, можно обеспечить большую гибкость будущих версий класса в отношении выполнения дополнительной работы в точке доступа свойства без какого-либо влияния на клиента. Например, предположим, что требуется обеспечить некоторое протоколирование в отладочной сборке при каждом обращении к свойству. Для доступа к данным клиент сможет неявно вызвать один из специальных методов свойств. Гибкость, которой можно достичь подобным образом, практически не ограничена. С другой стороны, при обращении к значению, представленному в общедоступном `readonly`-поле, никакой метод не вызывается, поэтому реализовать дополнительные действия можно, только перейдя от поля к свойству и перекомпилировав код.

Автореализуемые свойства

Очень часто возникает потребность в некотором типе (скажем, классе), который бы содержал несколько полей, объединенных в единую сущность. Например, представим тип `Employee`, содержащий полное имя и идентификационный номер, но для примера манипулирующий этими данными в виде строк, как показано ниже:

```
public class Employee
{
    string id;
    string fullName;
}
```

В том виде, как он написан, данный класс, по сути, бесполезен. Его два поля являются приватными, и к ним необходимо как-то открыть доступ. Чтобы сохранить инкапсуляцию, нельзя просто сделать эти поля общедоступными, и необходимо добавить значительный объём кода для создания двух связанных с этими полями свойств:

```
public class Employee
{
    string id;
    public string Id
    {
        get { return id; }
        set { id = value; }
    }

    string fullName;
    public string FullName
    {
        get { return fullName; }
        set { fullName = value; }
    }
}
```

К счастью, в C# 3.0 добавлено новое средство под названием *автореализуемые свойства* (*auto-implemented properties*), которое позволяет существенно сократить эту утомительную работу. Посмотрим, как изменится приведенный выше тип `Employee` за счет использования автореализуемых свойств:

```
public class Employee
{
    public string Id { get; set; }
    public string FullName { get; set; }
}
```

В этом примере мы сообщаем компилятору: «Мне нужно строковое свойство `FullName` с поддержкой для него средств доступа `get` и `set`». «За кулисами» компилятор генерирует в классе приватное поле для хранения значения и реализует средства доступа к нему. Если позднее понадобится каким-то образом настроить средство доступа к свойству `Id`, то это можно будет сделать без перекомпиляции кода клиентов `Employee`.

Можно также создать автореализуемое свойство, доступное только для чтения, вставив ключевое слово `private`:

```
public class Employee
{
    public string Id { get; private set; }
    public string FullName { get; set; }
}
```

Теперь свойство `Id` является доступным только для чтения, и задать его значение можно в конструкторе класса:

```
public class Employee
{
    public string Id { get; private set; }
    public string FullName { get; set; }

    public Employee(string id, string fullName)
    {
        Id = id;
        FullName = fullName;
    }
}

...
```

```
Employee emp = new Employee("111-11-1111", "John Doe");
```

Инкапсуляция

Инкапсуляцию можно считать одной из наиболее важных концепций объектно-ориентированного программирования. Инкапсуляция — это дисциплина тщательного контроля доступа к внутренним данным и процедурам объектов. Ни один язык, не поддерживающий инкапсуляцию, не может претендовать на звание объектно- ориентированного.

Всегда необходимо стараться следовать базовой концепции: никогда не определять поля данных объектов с открытым доступом. Необходимо сделать так, чтобы клиенты объекта общались с ним только управляемым образом. Обычно это означает организацию взаимодействия с объектом только через методы этого объекта (или свойства, которые, по сути, являются вызовами методов). Таким образом, внутренности объекта рассматриваются как «чёрный ящик». Внутреннее хозяйство объекта внешнему миру не видно, а все коммуникации, которые могут модифицировать внутренности, осуществляются по контролируемым каналам. С помощью инкапсуляции можно спроектировать такой дизайн, который гарантирует, что внутреннее состояние объекта никогда не будет нарушено.

Простой иллюстрацией к сказанному может быть следующий пример. Создадим фиктивный вспомогательный объект, представляющий прямоугольник:

```
class MyRectangle
{
    public uint width;
    public uint height;
}
```

В этом примитивном примере для прямоугольника определены только ширина и длина. Поэтому два поля — `width` и `height` — объявлены общедоступными. Возможно, это вызвано спешкой при проектировании этого базового маленького класса.

Теперь предположим, что со временем этот маленький класс применялся во многих местах. Не стоит забывать, что этот класс прямоугольника в том виде, как он есть, не слишком полезен, и потому пусть было решено сделать его более полезным. Предположим, что существует клиентский код, который использует этот класс прямоугольника и которому нужно вычислить площадь прямоугольника. Во времена ANSI C и других простых процедурных императивных языков понадобилось бы создать функцию с именем вроде `ComputeArea`, которая принимала бы параметр — указатель на экземпляр `MyRectangle`. Строгие принципы объектно-ориентированного программирования склоняют к тому, что лучший способ сделать это — позволить экземплярам `MyRectangle` самостоятельно сообщать клиенту значения площади:

```
class MyRectangle
{
    public uint width;
    public uint height;

    public uint GetArea()
    {
        return width * height;
    }
}
```

Как видите, в класс добавлен новый член — метод `GetArea`, при вызове которого на экземпляре класса `MyRectangle` он вычислит значение площади прямоугольника и вернёт результат. Предположим, что возникла причина предварительно вычислить площадь прямоугольника, чтобы каждый раз при вызове метода `GetArea` не пришлось вычислять её заново. Возможно, так нужно сделать потому, что известно, что `GetArea` будет вы-

зываться многократно с одним и тем же экземпляром на протяжении всего времени его существования. Игнорируя неразумность ранней оптимизации, представим, что это все-таки сделано. Теперь класс `MyRectangle` выглядит примерно так:

```
class MyRectangle
{
    public uint width;
    public uint height;
    public uint area;

    public uint GetArea()
    {
        return area;
    }
}
```

Такое описание класса может привести к ошибкам. Обратите внимание, что все поля являются общедоступными. Это позволяет потребителю экземпляров класса `MyRectangle` иметь прямой доступ к его внутренностям. Какой смысл в предоставлении метода `GetArea`, если потребитель может напрямую обратиться к полю `area`? Возможно, стоило бы сделать поле `area` приватным. Таким образом, для получения площади прямоугольника клиенты были бы вынуждены вызывать `GetArea`:

```
class MyRectangle
{
    public uint width;
    public uint height;

    private uint area;
    public uint GetArea()
    {
        if (area == 0)
            area = width * height;
        return area;
    }
}
```

Поле `area` сделано приватным, заставляя потребителя вызывать метод `GetArea`, чтобы получить площадь прямоугольника. Однако во время работы над примером стало ясно, что в какой-то момент должна быть вычислена площадь прямоугольника. По причине лени было решено проверять значение поля `area` перед возвратом, и если оно равно 0, то это значит, что оно должно быть предварительно вычислено. Это — грубая попытка оптимизации, но теперь площадь вычисляется только в случае необходимости. Предположим, что потребитель экземпляра прямоугольника никогда не пожелает узнать его площадь. Тогда, при условии использования предыдущего кода, ему никогда не придется тратить время на вычисление этой площади. Разумеется, в этом несколько надуманном примере оптимизация подобного рода будет весьма незначительной. Но если вы немного задумаетесь, то наверняка представите пример, который существенно выигрывает от применения отложенного вычисления. Например, при организации доступа к базе данных по медленной сети, когда во время выполнения нужны лишь несколько полей таблицы. Или для того же объекта доступа к базе данных может оказаться слишком дорогой операция подсчёта строк в таблице. Поэтому при необходимости можно применять описанный выше подход.

В классе прямоугольника все еще присутствует существенная проблема. Поскольку поля `width` и `height` являются общедоступными, что произойдет, если пользователь изменит одно из значений после вызова `GetArea` на экземпляре? В этом случае будет получен наихудший пример несогласованности внутренностей объекта. Целостность состояния объекта будет нарушена. Это определённо нехорошая ситуация. Таким образом, в коде по-прежнему присутствует ошибка. В связи с этим, поля `width` и `height` также должны быть сделаны приватными:

```
class MyRectangle
```

```
{
    private uint width;
    public uint Width
    {
        get { return width; }
        set
        {
            width = value;
            ComputeArea();
        }
    }

    private uint height;
    public uint Height
    {

        get { return height; }
        set
        {
            height = value;
            ComputeArea();
        }
    }

    private uint area;
    public uint Area
    {
        get { return area; }
    }
    private void ComputeArea()
    {
        area = width * height;
    }
}
```

Эта версия класса `MyRectangle` выглядит намного лучше. После того, как поля `width` и `height` были сделаны приватными, стало ясно, что потребителю этих объектов нужен какой-нибудь способ установки и получения значений ширины и высоты. И здесь пригодятся свойства C#. Теперь изменения внутреннего состояния будут обрабатываться в теле метода, а вызываемые при этом методы относятся к набору специально именованных методов класса. Теперь появилась возможность более строгого контроля доступа к внутренностям объекта. И вместе с этим контролем пришла более высокая степень инкапсуляции. Можно эффективно управлять внутренним состоянием объекта, так что оно никогда не станет несогласованным. Гарантировать целостность состояния объекта, когда внешние сущности имеют доступ к внутреннему состоянию «с черного хода», нельзя.

В данном примере объекту точно известно, когда изменяются поля `width` и `height`, поэтому он может предпринять необходимые действия для вычисления новой площади. Если в объекте используется подход с отложенным вычислением, сохраняя значение площади, вычисленное при первом чтении свойства `Area`, то значение кэша должно быть объявлено недействительным, как только будет вызван блок `set` любого из свойств `Width` или `Height`.

Таким образом, весьма незначительный объём дополнительной работы по обеспечению инкапсуляции со временем многократно окупается. Одно из самых замечательных свойств инкапсуляции, которое следует хорошо запомнить, состоит в том, что при использовании свойств внутреннее состояние объекта может изменяться для поддержки другого алгоритма, что никак не затронет его потребителей. Другими словами, интерфейс, видимый потребителю (также называемый контрактом), не изменяется. Например, в финальной реализации класса `MyRectangle` площадь вычисляется заново, как только устанавливается новое значение свойства `Width` или `Height`. Если позже выяснится, что предварительное вычисление площади сильно загружает процессор при работе приложения, то можно изменить модель, ориентируя её на применение кэшированного значения площади, которое вычисляется только при первом обращении, и поскольку соблюдались принципы инкап-

суляции, потребителям объектов даже не нужно знать об этом. Они не будут знать о том, что внутренняя реализация объекта изменилась. В этом и состоит мощь инкапсуляции. Когда изменяется внутренняя реализация объекта, а использующие его клиенты могут не меняться — значит, инкапсуляция работает так, как должна.

Инициализаторы объектов

В C# предусмотрено сокращение, которое можно использовать при инициализации новых экземпляров объектов. Обычно часто приходится писать код, подобный следующему:

```
Employee developer = new Employee();
developer.Name = "Fred Blaze";
developer.OfficeLocation = "B1";
```

Сразу после создания экземпляра `Employee` немедленно выполняется инициализация доступных свойств экземпляра. Было бы удобнее, если эти действия можно было бы выполнять в одном операторе. Для этого в C# существует возможность выполнения инициализатора объекта:

```
public class Employee
{
    public string Name { get; set; }
    public string OfficeLocation { get; set; }
}

...

Employee developer = new Employee
{
    Name = "Fred Blaze",
    OfficeLocation = "B1"
};
```

Обратите внимание на то, как инициализируется экземпляр `developer`. «За кулисами» компилятор генерирует тот же код, какой получился бы в случае, если бы свойства инициализировались вручную после создания экземпляра `Employee`. Поэтому такая техника работает только в случае доступности свойств — в данном случае `Name` и `OfficeLocation` — в точке инициализации.

Инициализаторы объектов можно даже вкладывать друг в друга, как показано в следующем примере:

```
public class Employee
{
    public string Name { get; set; }
    public string OfficeLocation { get; set; }
}

public class FeatureDevPair
{
    public Employee Developer { get; set; }
    public Employee QaEngineer { get; set; }
}

...

FeatureDevPair spellCheckerTeam = new FeatureDevPair
{
    Developer = new Employee
    {
        Name = "Fred Blaze",
        OfficeLocation = "B1"
    },
    QaEngineer = new Employee
    {
        Name = "Marisa Bozza",
        OfficeLocation = "L42"
```

```
    }
};
```

Обратите внимание, что два свойства объекта `spellCheckerTeam` инициализируются с использованием нового синтаксиса. Каждый экземпляр `Employee`, присваиваемый свойствам, сам инициализируется посредством объектного инициализатора. И наконец — ещё более краткий способ инициализации объектов, позволяющий сократить объём клавишного ввода за счёт скрытого усложнения:

```
public class Employee
{
    public string Name { get; set; }
    public string OfficeLocation { get; set; }
}

public class FeatureDevPair
{
    private Employee developer = new Employee();
    public Employee Developer
    {
        get { return developer; }
        set { developer = value; }
    }

    private Employee qaEngineer = new Employee();
    public Employee QaEngineer
    {
        get { return qaEngineer; }
        set { qaEngineer = value; }
    }
}

...

FeatureDevPair spellCheckerTeam = new FeatureDevPair
{
    Developer =
    {
        Name = "Fred Blaze",
        OfficeLocation = "B1"
    },
    QaEngineer =
    {
        Name = "Marisa Bozza",
        OfficeLocation = "L42"
    }
};
```

Как видите, здесь появилась возможность опустить выражения `new` при инициализации свойств `Developer` и `QaEngineer` объекта `spellCheckerTeam`. Однако такой сокращённый синтаксис требует, чтобы поля объекта `spellCheckerTeam` уже существовали перед установкой свойств, т.е. данные поля не должны быть `null`. В связи с этим пришлось изменить определение `FeatureDevPair`, чтобы создать содержащиеся экземпляры `Employee` в точке инициализации. Однако, такой подход требует внимательности, поскольку компилятор не может выявить ошибку, если значение объекта будет равно `null`, и во время выполнения программы будет сгенерировано исключение.

Класс `object`

Каждый объект в CLR наследуется от `System.Object` — базового типа для всех других типов. В C# ключевое слово `object` представляет собой псевдоним `System.Object`. То, что каждый тип в CLR и C# наследуется от класса `Object`, может оказаться удобным. Например, массив элементов разных типов можно трактовать как однородную, приведя их все к ссылкам на `object`.

Даже System.ValueType наследуется от Object. Однако получение ссылки на Object регулируется некоторыми специальными правилами. На ссылочных типах можно преобразовывать ссылку на класс A в ссылку на класс Object простым неявным преобразованием. Проход в обратном направлении требует проверки типа во время выполнения и явного приведения с использованием хорошо знакомого синтаксиса — предваряя преобразуемый экземпляр именем нового типа в скобках. Прямое получение ссылки Object для типа значения формально невозможно. Семантически это оправдано, поскольку типы значений расположены в стеке. Было бы опасно получить ссылку на кратковременно существующий экземпляр значения и сохранить её для последующего использования, когда этот экземпляр значения уже, вероятно, исчезнет. По этой причине получение ссылки на Object для типов значений сопряжено с операцией упаковки.

Определение класса System.Object выглядит так:

```
public class Object
{
    public Object();
    public virtual void Finalize();
    public virtual bool Equals(object obj);
    public static bool Equals(object obj1, object obj2);
    public virtual int GetHashCode();
    public Type GetType();
    protected object MemberwiseClone();
    public static bool ReferenceEquals(object obj1, object obj2);
    public virtual string ToString();
}
```

Object предоставляет ряд методов, которые проектировщики CLI/CLR сочли важными и подходящими для каждого объекта. В Object предусмотрен метод GetType для получения типа времени выполнения любого объекта, работающего в CLR. Возможность определения типов в системе во время выполнения в сочетании с рефлексией чрезвычайно удобна. Метод GetType возвращает объект типа Type, который представляет реальный, или конкретный, тип объекта. Используя этот возвращённый объект, можно узнать всё о типе объекта, с которым был вызван метод GetType. К тому же, имея две ссылки на тип Object, можно сравнивать результат вызова GetType для них обоих, чтобы узнать, являются ли они экземплярами одного и того же типа.

System.Object содержит метод по имени MemberwiseClone, возвращающий неглубокую копию объекта. Когда метод MemberwiseClone создает копию, все поля типов значений копируются бит за битом, в то время как все поля ссылочного типа просто копируются, так что и оригинал, и копия содержат ссылки на один и тот же объект. Если требуется, чтобы также создавались копии объектов, на которые есть ссылки, необходимо использовать специальные механизмы.

Четыре из методов класса Object являются виртуальными, и если их реализация в Object по умолчанию не подходит, её можно переопределить. Метод ToString удобен для генерации текстового, читабельного для человека вывода и строкового представления объекта. Например, во время разработки может понадобиться возможность трассировки объекта в отладочном выводе во время выполнения. В таких случаях имеет смысл переопределить ToString, чтобы он предоставлял детальную информацию об объекте и его внутреннем состоянии. Версия ToString по умолчанию просто вызывает реализацию ToString объекта Type, возвращённого методом GetType, а результате возвращает только имя типа объекта.

Метод Finalize имеет специальное назначение. В C# его явное переопределение не разрешено. Также нельзя его вызывать на объекте. Если необходимо переопределить этот метод в классе, можно воспользоваться синтаксисом деструктора C#.