8

Введение в объектно-ориентированное программирование

В ЭТОЙ ГЛАВЕ...

- Что такое объектно-ориентированное программирование
- Приемы ООП
- Связь приложений Windows Forms с ООП

В предыдущих разделах был изложен весь базовый материал по синтаксису языка C# и программированию на нем, а также показано, как отлаживать приложения и создавать полезные консольные приложения. Однако для доступа ко всей мощи C# и .NET Framework необходимо использовать *объектно-ориентированное программирование* (ООП). Вообще-то, как скоро станет понятно, вы уже немного знакомы с этой техникой, но для простоты мы не акцентировали на этом внимание.

В настоящей главе кодирование временно откладывается, а все внимание переносится на принципы, лежащие в основе объектно-ориентированного программирования. Вы увидите, что C# тесно связан с ООП. Все понятия, вводимые в этой главе, будут более подробно рассматриваться в последующих главах на конкретных примерах, так что не переживайте, если в каком-то материале не разберетесь сразу.

Сначала будут рассмотрены основы ООП и, конечно же, ответ на самый фундаментальный вопрос: что такое *объект*. Терминология ООП поначалу может показаться несколько запутанной, поэтому предоставляется много объяснений. Вы также увидите, что использование ООП требует другого взгляда на программирование.

Помимо общих принципов ООП, в главе затрагивается и область, которая требует глубокого понимания ООП — приложения Windows Forms. Такие приложения (работающие в среде Windows и содержащие меню, кнопки и т.п.) предоставляют огромную область для описания и позволяют продемонстрировать основные моменты ООП в среде Windows Forms.

Концепции объектно-ориентированного программирования, описанные в данной главе, обусловлены средой .NET, и некоторые из приводимых здесь приемов могут не работать в других объектно-ориентированных средах. При программировании на C# применяются приемы ООП из .NET, поэтому мы будем рассматривать именно их.

Что такое объектно-ориентированное программирование

Объектно-ориентированное программирование является относительно новым подходом к созданию компьютерных приложений, который призван устранить многие из проблем, существующих в традиционных методиках программирования. Вид программирования, с которым мы пока имели дело, называется *функциональным* (или *процедурным*) программированием и часто приводит к созданию так называемых монолитных приложений, все функции которых сконцентрированы в нескольких модулях кода (а то и вовсе в одном). В ООП обычно используется гораздо больше модулей, каждый из которых обеспечивает конкретные функции и может быть изолирован или даже полностью отделен от всех остальных. Такое модульное программирование обеспечивает гораздо большую гибкость и возможности для многократного использования кода.

Чтобы нагляднее показать, о чем идет речь, представьте, что высокопроизводительное компьютерное приложение является мощным гоночным автомобилем. При создании с помощью традиционных приемов программирования этот автомобиль будет представлять собой одно целое. Если понадобится что-то улучшить в этом автомобиле, его придется заменить целиком, т.е. отправить обратно изготовителю для переделки профессиональным механиком или вообще купить новый автомобиль. А технология ООП позволяет, например, купить у изготовителя новый двигатель и самостоятельно заменить его, следуя инструкциям изготовителя и не углубляясь в тонкости проведения ремонтных работ.

В традиционном приложении поток выполнения обычно прост и линеен. Приложения загружаются в память, начинают выполняться в точке А, завершают работу в точке Б и затем выгружаются из памяти. Попутно могут использоваться и другие разнообразные сущности, вроде файлов на носителе данных или возможностей видеокарты, но основная

часть обработки выполняется все-таки в одном месте. Сама обработка данных обычно несложная, использует различные математические и логические средства, а для построения более сложных представлений данных применяются простые типы вроде целочисленных или логических.

В ООП подобная линейность встречается редко. Результаты достигаются те же, но способ их получения зачастую выглядит совершенно по-другому. В ООП основной акцент делается на структуру и смысл данных, а также на взаимодействие этих данных с другими данными. Это обычно требует больше усилий на этапах проектирования приложения, но обеспечивает возможность его расширения. После принятия решения о представлении конкретных типов данных это представление может применяться в последующих версиях данного приложения и даже в совершенно новых приложениях. Наличие подобного соглашения может значительно сократить время разработки. Этим можно объяснить пример с гоночным автомобилем, где таким соглашением является структурирование кода "двигателя", который позволяет легко подставлять новый код (новый двигатель), не возвращая его изготовителю. Помимо этого, двигатель после создания можно использовать и для других целей — например, в другом автомобиле или вовсе в подводной лодке.

ООП часто упрощает программирование с помощью соглашений по представлению и применению более абстрактных объектов. Например, соглашение может приниматься не только по формату данных, используемых для отправки выходных данных на устройство вроде принтера, но и по методам обмена данными с этим устройством, т.е. инструкциям, которые оно должно понимать, и т.д. В автомобилестроении такое соглашение касалось бы способа подключения двигателя к топливной системе, трансмиссии и т.п.

Как видно из названия этой технологии, достигается это все с помощью *объектов*.

Что такое объект

Объект — это "строительный блок" в ООП-приложении. Такой строительный блок инкапсулирует часть приложения — процесс, порцию данных или какой-то более абстрактный объект.

Простейшие объекты могут быть очень похожи на знакомый нам тип структуры, который содержит члены типа переменных и функций. Содержащиеся переменные представляют хранимые в объекте данные, а содержащиеся функции обеспечивают доступ к возможным действиям этого объекта. Чуть более сложные объекты могут вообще не содержать данных, а представлять процесс и содержать только реализующие этот процесс функции. Примером такого объекта может служить принтер с функциями управления (распечатка документа, вывод тестовой страницы и т.д.).

Объекты в языке C# создаются из типов, как и хорошо знакомые составные переменные. Для типа объекта в ООП имеется специальное название — *класс*. Определения классов позволяют создавать объекты — т.е. реальные именованные *экземпляры* класса. Понятия "экземпляр класса" и "объект" эквивалентны, но термины "класс" и "объект" означают совершенно разные вещи.

Термины "класс" и "объект" часто путают, поэтому очень важно понимать, чем они отличаются. Помочь в этом может наша аналогия с гоночным автомобилем. Классом можно считать чертежи для изготовления автомобиля, а объектом — сам автомобиль, сделанный по этим чертежам.

В настоящей главе для работы с классами и объектами используется язык UML (Unified Modeling Language — унифицированный язык моделирования). Этот язык был специально разработан для моделирования программных систем — от объектов, из которых они состоят, и операций, которые они выполняют, до предполагаемых способов их использования. Здесь применяются и объясняются только базовые возможности языка UML, без более сложных аспектов, которым посвящены целые книги.

В VS имеется мощное средство для просмотра классов, которое позволяет отображать классы однотипным образом. Но для простоты все рисунки в данной главе выполнены вручную.

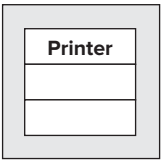

Рис. 8.1. *UML-представление класса* `Printer`

На рис. 8.1 для примера показано UML-представление класса принтера по имени `Printer`. Имя класса записано в самом верхнем разделе данного прямоугольника (о двух нижних разделах будет рассказано ниже).

На рис. 8.2 показано UML-представление экземпляра данного класса `Printer` по имени `myPrinter`.

Здесь в верхнем разделе записано имя экземпляра, а за ним, через двоеточие, имя его класса.

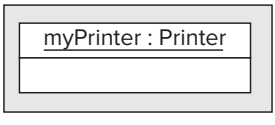

Рис. 8.2. *UML-представление экземпляра класса* `Printer` *по имени* `myPrinter`

Свойства и поля

Свойства и поля обеспечивают доступ к содержащимся в объекте данным. Эти данные как раз и являются тем, что отличает отдельные объекты друг от друга, поскольку в свойствах и полях разных объектов одного и того же класса могут храниться разные значения.

Различные фрагменты содержащихся в объекте данных вместе образуют *состояние* этого объекта. Например, пусть имеется класс объектов, представляющий чашку кофе и потому имеющий имя `CupOfCoffee`. При создании экземпляра (т.е. объекта) этого класса необходимо обеспечить его состояние, чтобы он был осмысленным. Для этого использующий данный объект код может с помощью свойств и полей задать сорт используемого кофе, содержится ли в кофе молоко и/или сахар, является ли кофе быстрорастворимым, и т.д. Тогда каждый конкретный объект `CupOfCofee` будет иметь определенное состояние, например: “Чашка колумбийского кофе с молоком и двумя кусочками сахара”.

Поля и свойства имеют типы, поэтому информация может в них храниться в виде значений `string`, `int` и т.д. Свойства отличаются от полей тем, что они не предоставляют непосредственный доступ к данным. Объекты могут ограждать пользователей от внутренних деталей своих данных, которые не обязательно взаимно однозначно представлены в существующих свойствах. Скажем, в поле для хранения информации о количестве кусочков сахара в экземпляре `CupOfCoffee` пользователи смогут помещать любые значения, ограниченные лишь пределами типа этого поля. То есть, например, в случае использования для хранения этих данных типа `int` пользователи смогут помещать в это поле любое значение в диапазоне от –2 147 483 648 до 2 147 483 647 (см. главу 3). Очевидно, что не все такие значения будут иметь смысл, особенно отрицательные, да и для слишком больших положительных может понадобиться чашка необычайно больших размеров. Использование свойства для хранения этой информации легко позволяет ограничить данное значение, скажем, только числами от 0 до 2.

В общем случае для доступа к состоянию лучше предоставлять свойства, а не поля, поскольку они позволяют управлять многими аспектами их поведения. Этот выбор никак не влияет на код, где применяются экземпляры объектов, т.к. синтаксис для использования свойств и полей выглядит одинаково.

Объект может четко определять и тип доступа к свойствам (для чтения и/или для записи). Некоторые свойства могут быть доступными только для чтения, т.е. они позволяют узнать их значения, но не изменять их (по крайней мере, непосредственно). Это часто бывает удобно для одновременного считывания нескольких фрагментов состояния. Например, класс `CupOfCoffee` может иметь доступное только для чтения свойство по имени `Description` (Описание), возвращающее строку с полной информацией о состоянии экземпляра этого класса. Конечно, эти сведения можно собрать и путем опроса нескольких свойств, но наличие такого свойства может сэкономить время и усилия. Аналогично ведут себя и свойства, доступные только для записи.

Кроме доступа для чтения и записи, для свойств и полей можно задавать разные виды *доступности*. Доступность членов класса определяет, какой код может получать доступ к этим членам, т.е. являются ли они доступными для всего кода (общедоступные), только для кода внутри класса (приватные) или следуют более сложной схеме (об этом будет рассказано позже). Как правило, поля делают приватными, а доступ к ним открывается через общедоступные свойства. При таком подходе код внутри класса имеет прямой доступ к хранящимся в поле данным, а общедоступное свойство ограждает внешних пользователей от этих данных и не позволяет им заносить туда недопустимое содержимое. Об общедоступных членах говорят, что они *предоставляются* классом.

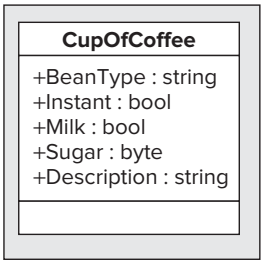

***Рис. 8.3.** Отображение имен свойств и полей в UML-представлении класса*

Доступность немного похожа на область видимости переменных. Например, приватные поля и свойства можно считать локальными для объекта, который ими владеет, а область видимости общедоступных полей и свойств охватывает и внешний для объекта код.

В UML-представлении класса имена свойств и полей отображаются во втором разделе, как показано на рис. 8.3.

На этом рисунке показано UML-представление класса `CupOfCoffee`, в котором определены пять членов (свойства или поля, поскольку в UML между ними нет никакой разницы). В каждой строке содержится следующая информация.

- Доступность. Общедоступные члены помечены символом "+", а приватные — символом "–". Правда, обычно в настоящей главе приватные члены не будут приводиться на диаграммах, поскольку эта информация является внутренней для класса. Доступ для чтения или записи не обозначается никак.
- Имя члена.
- Тип члена.

Имена членов и их типы разделяются двоеточием.

Методы

Термином *метод* принято называть предоставляемую объектом функцию. Методы могут вызываться так же, как и любые другие функции, и так же возвращать значения и принимать параметры (функции были рассмотрены в главе 6).

Методы применяются для доступа к функциональным возможностям объекта. Подобно полям и свойствам, они могут быть общедоступными или приватными, ограничивая при необходимости доступ к ним из внешнего кода. Нередко методы используют в своих действиях состояние объекта и обращаются к приватным членам. Например, в классе `CupOfCoffee` может быть определен метод `AddSugar()`, предоставляющий более понятный синтаксис для увеличения количества сахара, чем действия с соответствующим свойством `Sugar`.

В UML-представлении методы классов отображаются в третьем разделе прямоугольников, как показано на рис. 8.4.

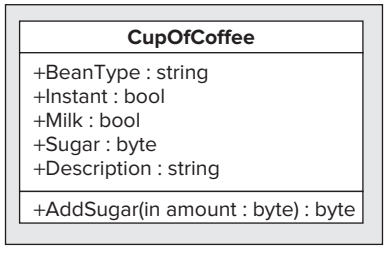

Рис. 8.4. *Отображение методов в UML-представлении класса*

Синтаксис записи методов в UML похож на синтаксис полей и свойств, только дополнительно записываются параметры методов, а тип в конце означает возвращаемый тип. Каждый параметр отображается в UML с одним из следующих идентификаторов: `in`, `out` или `inout`. Они обозначают направление потока данных, причем `out` и `inout` примерно соответствуют ключевым словам `out` и `ref` в языке C#, о которых рассказывалось в главе 6, а `in` — стандартному поведению в языке C#, когда не указаны ни `out`, ни `ref`.

Объектом является все, что угодно

Теперь пора внести ясность: объекты, свойства и методы уже не раз встречались в этой книге. Вообще-то в языке C# и .NET Framework объектом является все, что угодно. Функция `Main()` в консольном приложении является методом класса. Каждый из рассмотренных нами типов переменных является классом. Каждая из продемонстрированных команд, наподобие `<`*`строка`*`>.Length`, `<`*`строка`*`>.ToUpper()` и т.д. — это свойство или метод. (Символ точки здесь отделяет имя экземпляра объекта от имени свойства или метода; методы отличаются от свойств наличием скобок после них.)

Объекты встречаются повсюду, а синтаксис их использования обычно очень прост. До сих пор рассматривались лишь фундаментальные аспекты C#, и все приводимые примеры были достаточно простыми. С этого момента начнется более детальное изучение объектов. Очень важно запомнить, что описываемые здесь концепции имеют далеко идущие последствия — даже в отношении простой маленькой переменной `int`, с которой было так легко иметь дело.

Жизненный цикл объекта

Каждый объект имеет четко определенный жизненный цикл. Помимо обычного состояния — использование объекта — этот жизненный цикл включает два важных этапа.

- **Построение**. При создании экземпляра объекта его нужно инициализировать. Этот процесс инициализации и называется *построением*, и выполняется он функцией-конструктором, которую часто называют просто *конструктором*.
- **Уничтожение**. При уничтожении объекта часто требуется выполнить какие-либо операции по зачистке, вроде освобождения памяти. За их выполнение отвечает функция-деструктор, которую часто называют просто *деструктором*.

Конструкторы

Простая инициализация объекта осуществляется автоматически. Например, не нужно подыскивать место в памяти для размещения нового объекта. Однако иногда бывает необходимо выполнить на этапе инициализации объекта дополнительные задачи — например, инициализировать хранимые в объекте данные. Для этого применяется конструктор.

Все определения класса содержат, по крайней мере, один конструктор — *конструктор по умолчанию*, или *стандартный* конструктор — не принимающий параметров метод с таким же именем, как у самого класса. Определение класса может включать и несколько других конструкторов, принимающих параметры. Такие конструкторы позволяют создавать экземпляр объекта различными способами — например, предоставляя начальные значения для хранимых в объекте данных.

В языке C# конструкторы вызываются с помощью ключевого слова `new`. К примеру, создать экземпляр объекта `CupOfCoffee` с помощью конструктора по умолчанию можно следующим образом:

```
CupOfCoffee myCup = new CupOfCoffee();
```

Объекты можно создавать и с помощью конструкторов, отличных от конструктора по умолчанию. Например, у класса `CupOfCoffee` может существовать такой конструктор, принимающий параметр для указания типа кофейных зерен:

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

Конструкторы, подобно полям, свойствам и методам, могут быть общедоступными или приватными. Код, внешний по отношению к классу, не может создавать объекты с помощью приватного конструктора; он должен обязательно использовать общедоступный конструктор. Так можно, например, заставить пользователей классов применять конструктор, отличный от конструктора по умолчанию (сделав конструктор по умолчанию приватным).

Некоторые классы не имеют общедоступных конструкторов, т.е. создавать их экземпляры из внешнего кода невозможно (такие классы называются *не создаваемыми*). Однако, как будет показано ниже, это не означает, что они совсем бесполезны.

Деструкторы

Деструкторы используются в .NET Framework для выполнения очистки при удалении объектов. Обычно какой-то особый код для деструктора не требуется; все, что нужно, делается автоматически. Однако можно добавить и специальные действия, если перед удалением объекта необходимо выполнять какие-либо важные операции.

Например, после выхода переменной из области видимости она может быть не доступной из кода, но по-прежнему существовать где-то в памяти. Только при выполнении средой .NET сборки мусора такой экземпляр уничтожается полностью.

Не надейтесь на освобождение деструктором ресурсов, которые используются экземпляром объекта: это может произойти спустя много времени после того, как объект станет ненужным. Если данные ресурсы являются критичными, это чревато появлением проблем. Однако существует одно решение, которое рассматривается ниже в разделе "Освобождаемые объекты".

Статические члены классов и члены экземпляров классов

Помимо свойств, методов и полей, принадлежащих конкретным экземплярам объектов, могут существовать и *статические* (еще называемые *разделяемыми*, особенно в Visual Basic) члены, которые тоже могут быть методами, свойствами и полями. Статические члены совместно используются разными экземплярами класса и поэтому могут считаться глобальными для объектов конкретного класса. Статические свойства и поля позволяют обращаться к данным, которые не зависят ни от каких экземпляров объектов, а статические методы — выполнять команды, связанные с типом класса, но не с конкретными экземплярами объектов. Более того, для использования статических членов даже не нужно создавать объекты.

Например, статическими являются методы `Console.WriteLine()` и `Convert.ToString()`, которые уже демонстрировались ранее в книге. Создавать экземпляры классов `Console` и `Convert` для них не нужно (да это и невозможно, т.к. конструкторы этих классов не являются общедоступными, как описывалось ранее).

Существует еще много подобных ситуаций, в которых статические свойства и методы можно применять с пользой. Например, статическое свойство можно использовать для отслеживания количества созданных экземпляров класса. В UML-синтаксисе статические члены классов обозначаются подчеркиванием, как показано на рис. 8.5.

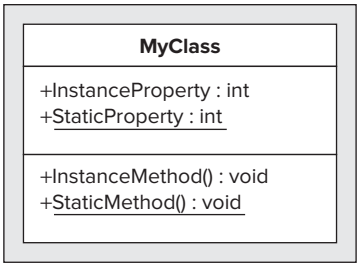

Рис. 8.5. *Статические члены в UML-представлении*

Статические конструкторы

Для применения статических членов класса может понадобиться сначала инициализировать их. Конечно, для статического члена можно указать начальное значение прямо в объявлении, но иногда нужна более сложная инициализация или, возможно, какие-то другие операции перед присваиванием значений или выполнением статических методов.

Для выполнения подобных задач инициализации предусмотрены статические конструкторы. У класса может быть только один статический конструктор, который не должен иметь модификаторов доступа и не может принимать никаких параметров. Статический конструктор вызывается не непосредственно, а в одном из следующих случаев:

- при создании экземпляра класса, содержащего данный статический конструктор;
- при обращении к статическому члену класса, содержащему данный статический конструктор.

В обоих случаях сначала вызывается статический конструктор, и только потом создается экземпляр класса или выполняется обращение к статическим членам. Сколько бы экземпляров класса ни создавалось, его статический конструктор вызывается только один раз. Чтобы отличать статические конструкторы от описанных ранее в этой главе, все не статические конструкторы называются также *конструкторами экземпляров*.

Статические классы

Часто бывает нужно использовать классы, содержащие только статические члены, для которых невозможно создавать объекты (вроде класса `Console`). Проще всего не делать все конструкторы класса приватными, а использовать *статический класс*. Статический класс может содержать только статические члены и по своей сути не может содержать конструкторы экземпляров. Однако в нем могут быть статические конструкторы, как было сказано в предыдущем разделе.

Новичкам в ООП не помешает сделать паузу перед изучением остального материала этой главы. Очень важно полностью разобраться со всеми базовыми понятиями и только затем приступать к рассмотрению более сложных аспектов.

Приемы объектно-ориентированного программирования

Теперь, когда изучены основы и известно, что такое объекты и как они работают, можно переходить к рассмотрению других функциональных возможностей объектов. В данном разделе речь пойдет о следующем:

- интерфейсы;
- наследование;
- полиморфизм;
- отношения между объектами;
- перегрузка операций;
- события;
- ссылочные типы и типы-значения.

Интерфейсы

Интерфейс (interface) — это коллекция общедоступных (а значит, не статических) методов и свойств, которые сгруппированы для инкапсуляции конкретной функциональности. После определения интерфейса его можно реализовать в классе. Это означает, что в таком случае класс будет поддерживать все свойства и члены, указанные данным интерфейсом.

Интерфейсы не существуют сами по себе. "Создать экземпляр" интерфейса, как для класса, нельзя. Кроме того, интерфейсы не могут содержать код с реализацией их членов, они могут лишь определять их. Реализация членов осуществляется в классах, реализующих данный интерфейс.

В приведенном ранее примере класса `CupOfCoffee` многие свойства и методы более общего назначения, вроде `AddSugar()`, `Milk`, `Sugar` и `Instant`, можно сгруппировать интерфейс с именем, скажем, `IHotDrink` (имена интерфейсов обычно начинаются с заглавной буквы `I`). Такой интерфейс можно применять для других объектов, например, объектов класса `CupOfTea`. Это позволило бы однотипно работать со всеми подобными объектами, хотя они могут иметь и собственные свойства (например, объекты `CupOfCoffee` — свойство `BeanType`, а объекты `CupOfTea` — свойство `LeafType`).

Интерфейсы, реализуемые для классов, изображаются в UML с помощью *кружочков*. На рис. 8.6 классы, реализующие интерфейс `IHotDrink`, вынесены в отдельный прямоугольник.

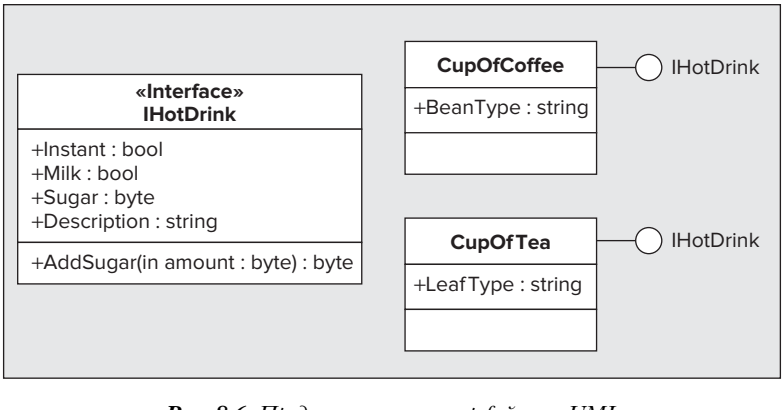

Рис. 8.6. *Представление интерфейсов в UML*

Один класс может поддерживать несколько интерфейсов, а несколько классов могут поддерживать один и тот же интерфейс. Значит, интерфейсы упрощают жизнь для пользователей и других разработчиков. Например, предположим, что имеется код, в котором используется объект с определенным интерфейсом. Если не использовать другие свойства и методы этого объекта, один объект можно будет легко заменять другим (код из примера с интерфейсом `IHotDrink`, например, может работать как с экземплярами `CupOfCoffee`, так и с экземплярами `CupOfTea`). Кроме того, разработчик данного класса сам может выпустить его обновленную версию, и если она поддерживает уже используемый интерфейс, другой разработчик сможет легко применить новую версию в своем коде.

После публикации интерфейса, т.е. предоставления к нему доступа другим разработчикам или конечным пользователям, лучше не изменять его. Интерфейс можно считать своего рода контрактом между создателями класса и его потребителями, в котором создатели заявляют, что каждый класс, реализующий интерфейс `X`, будет поддерживать такие-то методы и свойства. Если интерфейс позже изменится — например, из-за обновления базового кода — экземпляры классов будут работать неправильно или вообще не смогут работать. Поэтому вместо этого рекомендуется создавать новый интерфейс, расширяющий возможности старого и имеющий другой номер версии, вроде `X2`. Такой подход уже стал стандартным, поэтому интерфейсы с пронумерованными версиями встречаются довольно часто.

Освобождаемые объекты

Один из интерфейсов особенно интересен — это интерфейс `IDisposable`. Объект, поддерживающий этот интерфейс, должен реализовать метод `Dispose()`, т.е. предоставить код для этого метода. `Dispose()` может вызываться тогда, когда объект уже больше не нужен (например, непосредственно перед его выходом за пределы области видимости), и должен использоваться для освобождения любых критических ресурсов, которые в противном случае могут оставаться занятыми вплоть до вызова метода деструктора во время сборки мусора. Это обеспечивает больший контроль над ресурсами, которыми пользуются объекты.

В языке C# имеется конструкция специально для эффективного применения этого метода. Ключевое слово `using` позволяет инициализировать использующий критические ресурсы объект в кодовом блоке, при достижении конца которого автоматически вызывается метод `Dispose()`:

```
<имяКласса> <имяПеременной> = new <имяКласса>();
...
using (<имяПеременной>)
{
  ...
}
```

Объект *<имяПеременной>* можно создать и в составе оператора `using`:

```
using (<имяКласса> <имяПеременной> = new <имяКласса>)
{
  ...
}
```

И в том, и в другом случае переменная *<имяПеременной>* будет доступна внутри блока `using` и автоматически удалена в его конце (т.е. по завершении выполнения кода этого блока будет автоматически вызван метод `Dispose()`).

Наследование

Наследование (inheritance) — один из самых важных механизмов в ООП. Любой класс может наследоваться от другого класса, а это значит, что он будет иметь все те члены, что и класс, от которого он унаследован. В терминологии ООП класс, *от* которого наследуется

(порождается) другой класс, называется *родительским* или *базовым* классом. Классы в C# могут непосредственно наследоваться только от одного базового класса, хотя у того базового класса может быть собственный базовый класс, и т.д.

Механизм наследования позволяет расширять или создавать конкретные классы от одного более общего базового класса. Например, возьмем класс, представляющий животное с фермы. Этот класс мог бы называться `Animal` (Животное) и обладать методами вроде `EatFood()` (Питаться) или `Breed()` (Плодиться). От него можно создать производный класс по имени `Cow` (Корова), который поддерживает все эти методы, но при этом имеет и собственные — например, `Moo()` (Мычать) и `SupplyMilk()` (Давать молоко). Другим производным классом может быть класс `Chicken` (Курица) с методами `Cluck()` (Кудахтать) и `LayEgg()` (Снести яйцо).

В UML наследование изображается стрелками, как показано на рис. 8.7.

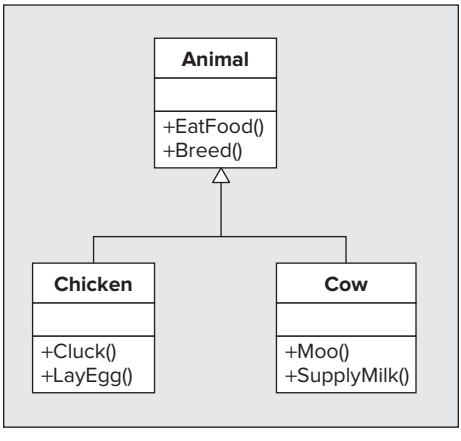

Рис. 8.7. *Представление отношений наследования в UML*

На рис. 8.7 возвращаемые типы членов для простоты не показаны.

При порождении от базового класса становится важным вопрос о доступности членов. Приватные члены базового класса недоступны из производного класса, а общедоступные (естественно) доступны. Но общедоступные члены доступны не только из производного класса, но и из внешнего кода. Поэтому если использовать только два этих уровня доступности, то невозможно создавать члены, доступные как из базового, так и из производного класса, но не из внешнего кода.

Для обхода этой проблемы существует третий уровень доступности — `protected` (защищенный), при котором доступ к члену имеют только производные классы. Для внешнего кода уровень доступности `protected` идентичен уровню `private`.

Помимо уровня защиты, для члена можно определить и способ наследования. Члены базового класса могут быть *виртуальными* (virtual), а это означает то, что они могут переопределяться в наследующем их классе, т.е. производный класс может содержать альтернативную реализацию таких членов. Эта альтернативная реализация не отменяет исходной реализации, которая все так же доступна в рамках исходного класса, но она закрывает ее от внешнего кода. При отсутствии альтернативной реализации любой внешний код, обращающийся к члену через производный класс, автоматически использует реализацию этого члена из базового класса.

Виртуальные члены не могут быть приватными: ведь невозможно, чтобы член мог переопределяться в производном классе и в то же время не был доступен из этого производного класса.

В примере с классом `Animal` можно сделать виртуальным метод `EatFood()` и предоставить для него новую реализацию в каком-то производном классе, например, в `Cow`, как показано на рис. 8.8. На этом рисунке метод `EatFood()` отображается и в классе `Animal`, и в классе `Cow`; это указывает на то, что у каждого из классов имеется собственная реализация данного метода.

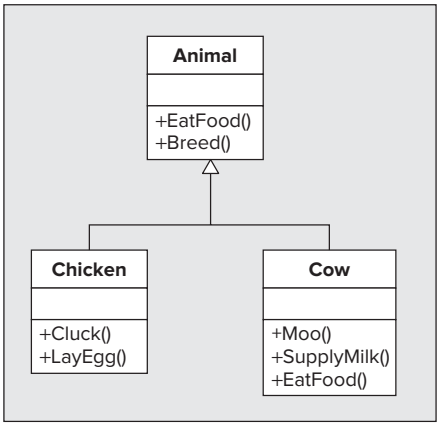

Рис. 8.8. *Представление виртуального метода в UML*

Базовые классы могут также определяться как *абстрактные* (abstract). Создавать экземпляр абстрактного класса непосредственно нельзя; использовать такой класс можно только для создания порожденных классов. У абстрактных классов могут быть абстрактные члены, которые не могут иметь реализацию в базовом классе — она должна быть представлена в производных классах. Например, если бы класс `Animal` был абстрактным, то в UML-представлении он выглядел бы так, как показано на рис. 8.9.

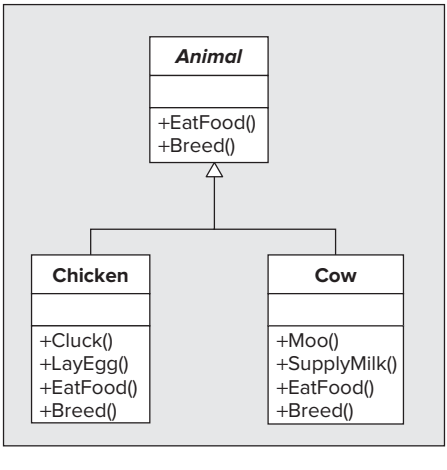

Рис. 8.9. *Представление абстрактных классов в UML*

Имена абстрактных классов записываются в UML курсивом (или выделяются пунктирной рамкой).

На рис. 8.9 методы `EatFood()` и `Breed()` указаны в абстрактных классах `Chicken` и `Cow`; это значит, что они являются либо абстрактными (и тогда подлежат переопределению в производных классах), либо виртуальными (т.е. уже определены в самих классах `Chicken` и `Cow`). Разумеется, абстрактные базовые классы могут содержать реализации для своих членов, и так нередко и бывает. Невозможность создания экземпляра абстрактного класса не означает, что в нем нельзя инкапсулировать функциональные возможности.

И, наконец, классы могут быть *запечатанными* (sealed). Такие классы не могут выступать в роли базового класса и, следовательно, не могут иметь производных классов.

В языке C# имеется один общий базовый класс для всех объектов, имеющий имя `object` (это псевдоним для класса `System.Object` из .NET Framework). Этот класс более подробно рассматривается в главе 9.

Интерфейсы, о которых рассказывалось в этой главе, тоже могут наследоваться от других интерфейсов. Но, в отличие от классов, они могут наследоваться от нескольких базовых интерфейсов (так же, как и классы могут поддерживать несколько интерфейсов).

Полиморфизм

Одним из результатов наследования является наличие в производных классах методов и свойств, совпадающих с базовым классом. Поэтому для экземпляров классов с общим базовым типом часто возможно применять идентичный синтаксис. Например, при наличии у класса `Animal` метода `EatFood()` синтаксис для вызова этого метода из производных классов `Cow` и `Chicken` будет одинаковым:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

Полиморфизм (polymorphism) позволяет двинуться еще дальше: присваивать значение переменной производного типа переменной базового типа:

```
Animal myAnimal = myCow;
```

Приведение при этом не требуется. Далее можно просто вызывать методы базового класса через эту переменную:

```
myAnimal.EatFood();
```

Данная строка кода приведет к вызову реализации метода `EatFood()` из производного класса. Однако вызывать подобным образом методы, определенные в производном классе, нельзя. То есть следующий код работать не будет:

```
myAnimal.Moo();
```

Хотя можно привести переменную типа базового класса к типу производного класса и вызвать метод производного класса:

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

Эта операция приведения приведет к исключению, если тип исходной переменной не совпадает ни с типом `Cow`, ни с типом производного от него класса. Для определения типа объекта существуют свои приемы, которые будут описаны в следующей главе.

Полиморфизм чрезвычайно полезен тем, что минимизирует объем кода для выполнения действий с разными объектами, происходящими от одного класса. Полиморфизм может применяться не только в отношении классов, имеющих одинаковый родительский класс. Он может применяться и, скажем, в отношении дочерних и "внучатых" классов — главное, чтобы в их иерархии наследования имелся общий класс.

Еще раз напоминаем, что в C# все классы порождаются от базового класса `object`, который является корневым в их иерархиях наследования. Поэтому все объекты можно считать экземплярами класса `object`. Это и позволяет методу `Console.WriteLine()` при построении строк обрабатывать практически бесконечное количество комбинаций параметров. Каждый параметр после первого считается экземпляром `object`, что позволяет выводить данные любого объекта. Для этого вызывается метод `ToString()`, являющийся членом класса `object`. Этот метод можно переопределить более подходящей реализацией, а можно просто воспользоваться стандартной реализацией, в которой он возвращает имя класса (уточненное пространствами имен, в которых он присутствует).

Полиморфизм интерфейсов

Создавать экземпляры интерфейсов таким же образом, как и экземпляры объектов, нельзя, но можно создать переменную типа интерфейса и затем использовать ее для обращения к методам и свойствам, предоставляемым этим интерфейсом в объектах, которые его поддерживают.

Например, предположим, что вместо базового класса `Animal` метод `EatFood()` помещен в интерфейс по имени `IConsume`. Этот интерфейс могут поддерживать оба класса — `Cow` и `Chicken`, — только каждый из них должен иметь свою реализацию метода `EatFood()`, поскольку интерфейсы не содержат реализаций. После этого к данному методу можно обращаться с помощью примерно такого кода:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

Получается простой способ для однотипного вызова различных методов, который не зависит от общего базового класса. Например, такой интерфейс можно реализовать и в классе `VenusFlyTrap` (Мухоловка), порожденном не от класса `Animal` (Животное), а от класса `Vegetable` (Растение):

```
VenusFlyTrapmyVenusFlyTrap=newVenusFlyTrap();
IConsumeconsumeInterface;
consumeInterface=myVenusFlyTrap;
consumeInterface.EatFood();
```

В этом коде вызов `consumeInterface.EatFood()` приводит к вызову метода `EatFood()` класса `Cow`, или `Chicken`, или `VenusFlyTrap` — в зависимости от того, какой экземпляр будет присвоен переменной типа интерфейса.

Производные классы наследуют интерфейсы, поддерживаемые их базовыми классами. В первом из предыдущих примеров интерфейс `IConsume` может поддерживаться как классом `Animal`, так и обоими классами `Cow` и `Chicken`. Учтите, что классы с общим базовым классом не обязательно имеют общие интерфейсы, и наоборот.

Отношения между объектами

Наследование является простым отношением между объектами, когда базовый класс полностью отражен производным классом, и производный класс может также иметь дос-

туп к внутренним деталям своего базового класса (через защищенные члены). Однако бывают и другие ситуации, в которых отношения между объектами более важны.

В настоящем разделе будут кратко рассмотрены следующие отношения.

- **Включение**. Один класс содержит другой. Такие отношения похожи на наследование, но позволяют содержащему классу управлять доступом к членам содержащегося внутри него класса и даже выполнять дополнительную обработку перед использованием этих членов.
- **Коллекции**. Один класс выступает в роли контейнера для нескольких экземпляров другого класса. Это похоже на массивы объектов, но у коллекций имеются и другие возможности: индексация, поиск, изменение размера и т.д.

Включение

Включение легко достигается использованием поля-члена для хранения экземпляра объекта. Это поле может быть общедоступным, и тогда пользователи содержащего объекта будут иметь доступ к предоставляемым им методам и свойствам — примерно как при наследовании. Но доступа к внутренним деталям класса через производный класс, который возможен в случае наследования, все-таки не будет.

Содержащийся внутри объект-член можно сделать приватным членом. В таком случае ни один из его членов не будет доступен пользователям непосредственно, даже если они являются общедоступными. Тогда обращаться к этим членам можно с помощью членов класса-контейнера. Такой подход позволяет полностью управлять тем, какие члены содержащегося внутри класса должны предоставляться (и должны ли вообще), а также выполнять дополнительную обработку в членах класса-контейнера перед обращением к членам содержащегося класса.

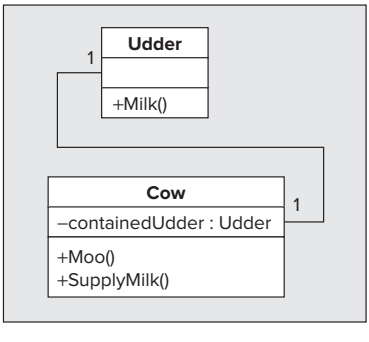

***Рис. 8.10.** Представление включения в UML*

Например, класс `Cow` может содержать класс `Udder` (Вымя) с общедоступным методом `Milk()` (Доить). Объект `Cow` может вызывать этот метод — например, в составе своего метода `SupplyMilk()`, но пользователям объекта `Cow()` такие детали не видны (и не важны).

Содержащиеся внутри других классы в UML могут изображаться с помощью связующих линий. В случае простого включения концы этих линий обозначаются единицами (1), что означает тип отношения один к одному (т.е., например, что один экземпляр `Cow` будет содержать один экземпляр `Udder`). Для большей наглядности экземпляр класса `Udder` может быть также изображен в виде приватного поля класса `Cow`, как показано на рис. 8.10.

Коллекции

В главе 5 были описаны массивы, предназначенные для хранения нескольких переменных одинакового типа. То же самое можно делать и для объектов (и неудивительно: вспомните, что уже знакомые вам типы переменных на самом деле являются объектами). Например:

```
Animal[] animals = new Animal[5];
```

Коллекция — это, по сути, массив, но с дополнительными возможностями. Коллекции реализуются в виде классов практически так же, как и другие объекты. Их имена часто представляют собой множественное число имени хранимых в них объектов: например, класс с именем `Animals` может содержать коллекцию объектов `Animal`.

Главное отличие от массивов состоит в том, что коллекции обычно реализуют дополнительные функции, вроде методов `Add()` и `Remove()` для добавления элементов в коллекцию и удаления из нее. У них также обычно имеется свойство `Item`, которое возвращает объект по его индексу. Довольно часто это свойство реализуется так, чтобы был возможен более сложный доступ. Например, класс `Animals` можно спроектировать так, чтобы обращаться к каждому конкретному объекту `Animal` по его имени.

В UML подобные отношения изображаются так, как показано на рис. 8.11.

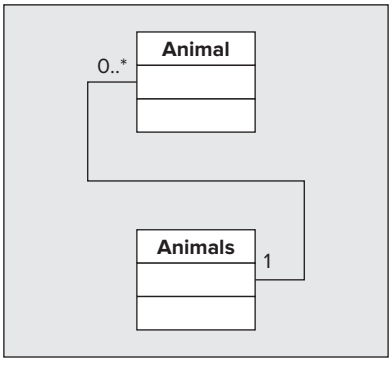

Рис. 8.11. *Представление коллекции в UML*

Члены на этом рисунке не показаны, т.к. здесь рассматриваются отношения. Числа на концах связующих линий указывают, что один объект `Animals` содержит ноль или более объектов `Animal`. Коллекции будут более подробно рассмотрены в главе 11.

Перегрузка операций

Вы уже знаете, как применять операции для обработки переменных простых типов. В некоторых случаях естественно использовать операции и с объектами собственных классов. Это возможно, потому что классы могут содержать инструкции по выполнению операций.

Например, в класс `Animal` можно добавить новое свойство `Weight` (Вес); тогда можно сравнивать вес животных с помощью такого кода:

```
if (cowA.Weight > cowB.Weight)
{
  ...
}
```

Перегрузка операций позволяет предоставить логику неявного использования свойства `Weight`, чтобы можно было написать так:

```
if (cowA > cowB)
{
  ...
}
```

Здесь операция “больше” (>) *перегружена*. Перегруженной называется такая операция, для которой был написан выполняющий ее код; этот код добавляется в определение одного из классов, для которого она должна выполняться. В предыдущем примере используются два объекта `Cow`, поэтому определение перегрузки операции содержится в классе `Cow`. Аналогично можно перегружать операции и для работы с разными классами — тогда соответствующий код должен находиться в одном из определений классов (или в обоих).

Перегрузка возможна только для операций, существующих в языке C#: новые операции создавать нельзя. Однако реализации можно предоставлять как для унарных, так и для бинарных версий операции вроде +. Как это сделать, будет рассказано в главе 13.

События

Объекты при их работе могут генерировать (и обрабатывать) *события* (event). События важны тем, что позволяют выполнять определенные действия в других частях кода — этим они похожи на исключения, но мощнее их. Например, при добавлении объекта `Animal` в коллекцию `Animals` может понадобиться выполнять определенный код, не являющийся ни частью класса `Animals`, ни частью того кода, который вызывает метод `Add()`. Для этого потребуется добавить в код *обработчик события* — особую функцию, которая вызывается при возникновении события. Кроме того, нужно настроить этот обработчик, чтобы он ожидал возникновения именно интересующего события.

С помощью событий можно создавать *управляемые событиями* приложения, которые гораздо более полезны, чем может показаться поначалу. Достаточно вспомнить, что, например, все Windows-приложения полностью зависят от событий. Каждый щелчок пользователя на кнопке или перетаскивание ползунка на полосе прокрутки выполняется с помощью обработки событий, генерируемых мышью или клавиатурой.

Как именно это все происходит в Windows-приложениях, будет показано ниже в этой главе, а вообще события детально рассматриваются в главе 13.

Ссылочные типы и типы значения

Данные в C# сохраняются в переменной одним из двух способов, который зависит от типа переменной. Этот тип относится к одной из двух категорий: ссылка или значение. Основные их отличия описаны ниже.

- Типы значения хранят себя и свое содержимое в одном месте в памяти.
- Ссылочные типы хранят ссылку на другое место в памяти (называемое *кучей* (heap)), в котором и хранится их содержимое.

Вообще-то при работе с C# особо беспокоиться об этом не нужно. Пока что в этой книге переменные типа `string` (ссылочный тип) и другие простые переменные (большинство из которых относится к типам значения, как, например, `int`) применялись практически одинаково.

Одно из главных отличий между типами значения и ссылочными типами состоит в том, что типы значения всегда содержат значения, а ссылочные типы могут быть пустыми (`null`), т.е. вообще не содержать значения. Хотя можно создать тип-значение, ведущий себя в этом отношении подобно ссылочному типу (может быть нулевым), путем использования *типов, допускающих нулевые значения* (nullable), которые являются разновидностью *обобщений* (generics). Обобщения представляют собой усовершенствованную технологию и более подробно рассматриваются в главе 12.

Единственными простыми ссылочными типами являются `string` и `object`, хотя неявно к ним относятся и массивы. Каждый создаваемый класс представляет собой ссылочный тип, поэтому здесь речь пойдет именно о них.

Главное отличие между структурами и классами состоит в том, что структуры представляют собой типы значения. Вы, наверно, обращали внимание на сходство структур и классов, особенно в главе 6, где было продемонстрировано использование функций в структурах. Более подробно об этом пойдет речь в главе 9.

Объектно-ориентированное программирование в Windows-приложениях

В главе 2 был приведен пример создания простого Windows-приложения на языке C#. Windows-приложения очень сильно зависят от ООП, и поэтому в настоящем разделе будут продемонстрированы некоторые его аспекты. В следующем практическом занятии предлагается еще один простой пример.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Объекты в действии

1. Создайте новое Windows-приложение с именем `Ch08Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter08`.
2. Добавьте в него новый элемент управления `Button` с панели Toolbox и поместите его в центр формы `Form1`, как показано на рис. 8.12.
3. Дважды щелкните на элементе `Button`, чтобы добавить код для обработки щелчка. Измените автоматически сгенерированный код следующим образом:

```
private void button1_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!";       // Выполнен щелчок
   Button newButton = new Button();
   newButton.Text = "New Button!";           // Новая кнопка!
   newButton.Click += new EventHandler(newButton_Click);
   Controls.Add(newButton);
}
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

Фрагмент кода ***Ch08Ex01\Form1.cs***

4. Запустите приложение. После этого на экране должна появиться форма, выглядящая так, как показано на рис. 8.13.
5. Щелкните на кнопке с надписью button1. Изображение на экране должно измениться (рис. 8.14).

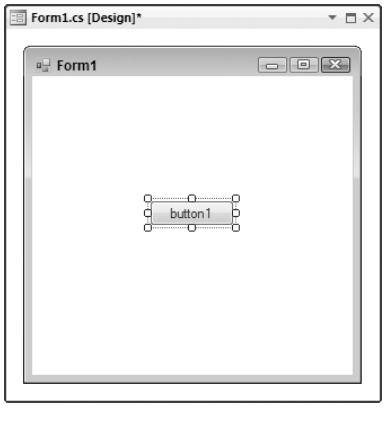

Рис. 8.12. *Окно с кнопкой*

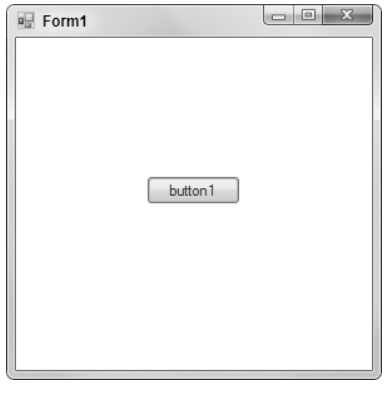

Рис. 8.13. *Приложения* `Ch08Ex01` *сразу после запуска*

6. Щелкните на кнопке с надписью New Button!. После этого изображение на экране должно измениться так, как показано на рис. 8.15.

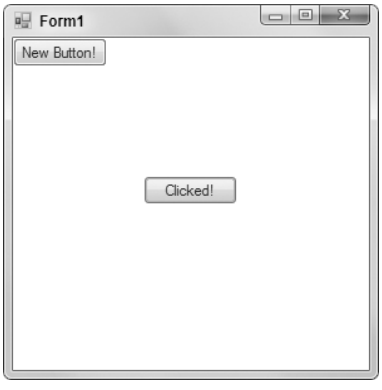

Рис. 8.14. *Приложение* `Ch08Ex01` *после щелчка на кнопке* **button1**

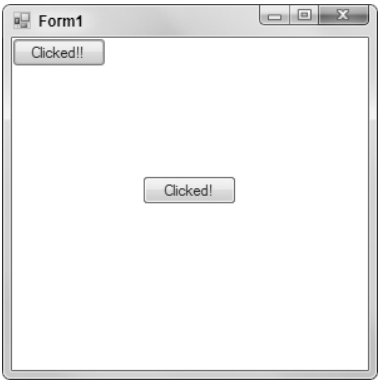

Рис. 8.15. *Приложение* `Ch08Ex01` *после щелчка на кнопке* **New Button!**

Описание работы

Добавление лишь нескольких строк кода позволило создать Windows-приложение, которое кое-что делает и демонстрирует применение некоторых приемов ООП в C#. Выражение "объектом является все, что угодно" в случае Windows-приложений справедливо еще более. Начиная с выполняемой формы и заканчивая элементами управления на этой форме — везде нужны приемы ООП. В данном упражнении были задействованы несколько рассмотренных в этой главе понятий, чтобы показать, как они работают все вместе.

Сначала на форму `Form1` приложения была добавлена новая кнопка. Кнопка представляет собой объект `Button`, а форма — объект `Form1`, порожденный от класса `Form`. Далее после двойного щелчка на кнопке был добавлен обработчик события для перехвата события `Click`, генерируемого объектом `Button`. Этот обработчик добавлен в код объекта `Form`, инкапсулирующего приложение, в виде приватного метода:

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

В этом коде используется спецификатор — ключевое слово `private`. Пока не обращайте на это внимания; код C#, необходимый для применения приемов ООП, описанных в настоящей главе, будет разъяснен в следующей главе.

В первой добавленной строке кода изменяется текст на кнопке, на которой выполняется щелчок, здесь задействуется полиморфизм, о котором рассказывалось ранее в главе. Объект `Button`, представляющий эту кнопку, отправляется обработчику событий через параметр `object`, который приводится к типу `Button` (это возможно, поскольку класс `Button` порожден от `System.Object`, т.е. класса .NET с псевдонимом `object`). Затем изменяется свойство `Text` этого объекта для изменения отображаемого текста:

```
((Button)sender).Text = "Clicked!";
```

Далее с помощью ключевого слова `new` создается новый объект `Button` (в этом проекте заданы пространства имен, позволяющие применять простой синтаксис; иначе для создания этого объекта пришлось бы использовать полностью квалифицированное имя `System.Windows.Forms.Button`):

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

Ниже в коде добавлен соответствующий новый обработчик событий, который используется для реагирования на событие `Click`, генерируемое новой кнопкой:

```
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

Затем этот обработчик регистрируется в качестве слушателя события `Click` с использованием синтаксиса перегрузки операций. С помощью конструктора создается новый объект `EventHandler` с именем новой функции-обработчика событий:

```
newButton.Click += new EventHandler(newButton_Click);
```

В конце используется свойство `Controls`. Это объект, представляющий коллекцию всех имеющихся на форме элементов управления, а его метод `Add()` применяется для добавления на форму новой кнопки:

```
Controls.Add(newButton);
```

Свойство `Controls` демонстрирует, что свойства не обязательно должны иметь простые типы вроде строк или целых чисел: они могут представлять собой любые объекты. В этом коротком примере были использованы почти все из описанных в настоящей главе приемов. Как видите, объектно-ориентированное программирование не обязательно должно быть сложным — это просто другой взгляд на программирование.

Резюме

В этой главе было приведено полное описание приемов объектно-ориентированного программирования — в контексте программирования на C#, но на принципы это не влияет. То есть большая часть материала данной главы описывает ООП на любом языке.

Сначала были рассмотрены основные вопросы: что означает термин “объект”, и как объект может быть экземпляром класса. Далее было рассказано, что объекты могут содержать разные члены — поля, свойства и методы, что эти члены могут иметь ограниченную доступность, и чем общедоступные члены отличаются от приватных. Потом вы узнали, что члены могут быть еще и защищенными, а также виртуальными и абстрактными (и что абстрактные методы возможны только в абстрактных классах). Вы также узнали, чем статические (совместно используемые) члены отличаются от членов экземпляров и почему иногда лучше использовать статические классы.

Затем был вкратце рассмотрен жизненный цикл объектов, наряду с их созданием конструкторами и удалением с помощью деструкторов. Позже, после рассмотрения группировки членов в интерфейсах, была описана и более сложная методика уничтожения объектов с помощью освобождаемых объектов, поддерживающих интерфейс `IDisposable`.

Остальная часть главы была в основном посвящена перечислению возможностей ООП, многие из которых будут более подробно рассмотрены в последующих главах. Вы познакомились с наследованием, позволяющим порождать классы от базовых классов, с двумя разновидностями полиморфизма (с помощью базовых классов и совместно используемые интерфейсов) и с включением в объект одного или более других объектов (с помощью включения и коллекций). И в конце было описано упрощение синтаксиса использования объектов с помощью перегрузки операций и то, как объекты генерируют события.

В заключительной части главы большая часть рассмотренных теоретических сведений была продемонстрирована на примере создания простого Windows-приложения. В следующей главе речь пойдет об определении классов в C#.

Упражнения

1. Какие из перечисленных ниже уровней доступности действительно существуют в ООП?

   а) `friend`

   б) `public`

   в) `secure`

   г) `private`

   д) `protected`

   е) `loose`

   ж) `wildcard`

2. "Деструктор объекта нужно обязательно вызывать вручную, иначе память будет использоваться неэффективно". Верно или нет?

3. Нужно ли создавать объект для вызова статического метода его класса?

4. Нарисуйте UML-диаграмму, подобную приведенным в главе, для перечисленных ниже классов и интерфейса.

   - Абстрактный класс с именем `HotDrink` (Горячий напиток), методами `Drink` (Пить), `AddMilk` (Добавить молока) и `AddSugar` (Добавить сахара) и свойствами `Milk` (Молоко) и `Sugar` (Сахар).
   - Интерфейс `ICup` (Чашка), содержащий методы `Refill` (Наполнить снова) и `Wash` (Помыть), а также свойства `Color` (Цвет) и `Volume` (Объем).
   - Класс с именем `CupOfCoffee` (Чашка кофе), порожденный от класса `HotDrink` (Горячий напиток), поддерживающий интерфейс `ICup` и обладающий дополнительным свойством `BeanType` (Сорт кофе).
   - Класс с именем `CupOfTea` (Чашка чая), порожденный от класса `HotDrink`, поддерживающий интерфейс `ICup` и обладающий дополнительным свойством `LeafType` (Сорт чая).

5. Напишите код для функции, которая в качестве параметра принимает один из двух возможных в предыдущем примере объектов типа `Cup...` (`CupOfCoffee` или `CupOfTea`). Эта функция должна вызывать методы `AddMilk`, `Drink` и `Wash` для любого передаваемого ей объекта `Cup`.

   Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Объекты и классы	Объекты — строительные блоки приложений на основе ООП. Классы — это определения типов, которые используются для создания объектов. Объекты могут содержать данные и/или предоставлять операции для выполнения в других частях кода. Данные можно сделать доступными для внешнего кода с помощью свойств, а операции — с помощью методов. Свойства и методы вместе называются членами класса. Свойства могут иметь разрешение на чтение, запись или и то, и другое. Члены класса могут быть общедоступными (доступными для всего кода) или приватными (доступными только в коде определения класса). В .NET все является объектом.
Жизненный цикл объекта	Объект создается с помощью вызова одного из конструкторов класса. Когда объект становится уже не нужным, он уничтожается с помощью деструктора. Для зачистки после использования объекта часто бывает необходимо избавляться от него вручную.
Статические члены и члены экземпляров	Члены экземпляров доступны только в объектах — экземплярах класса. Статические члены доступны только непосредственно через определение класса, они не связаны ни с каким экземпляром класса.
Интерфейсы	Интерфейс — это коллекция общедоступных свойств и методов, которые могут быть реализованы в классе. Переменной с типом экземпляра класса можно присвоить значение любого объекта, определение класса которого реализует этот интерфейс. Тогда через эту переменную будут доступны члены, определенную в данном интерфейсе.
Наследование	Наследование — это механизм порождения определения одного класса на основе другого. Дочерний класс наследует члены от (единственного) родительского класса. Дочерние классы не могут наследовать приватные члены своего родителя, но можно определить защищенные члены, которые доступны как внутри самого класса, так и внутри классов, порожденных от данного класса. Дочерние классы могут перекрывать члены, определенные в родительском классе как виртуальные. Цепочки наследования всех классов заканчиваются на `System.Object`, который в C# имеет псевдоним `object`.
Полиморфизм	Все объекты, созданные на основе порожденного класса, можно считать экземплярами родительского класса.
Отношения между объектами и дополнительные возможности	Объекты могут содержать другие объекты, а также могут представлять собой коллекции других объектов. Для работы с объектами в выражениях можно определить способ их обработки операциями с помощью перегрузки операций. Объекты могут предоставлять события, которые генерируются в каких-то внутренних процессах, а клиентский код может реагировать на события с помощью обработчиков событий.