5 Создание пользовательских типов с помощью объектно-ориентированного программирования

Данная глава посвящена созданию пользовательских типов с помощью принципов *объектно-ориентированного программирования (ООП)*. Вы узнаете о различных категориях элементов, которые может иметь тип, в том числе о полях для хранения данных и методах для выполнения действий. Вы будете применять концепции ООП, такие как агрегирование и инкапсуляция. Вы изучите языковые функции, такие как поддержка синтаксиса кортежей и переменные `out`, литералы для значений по умолчанию и автоматически определяемые имена кортежей.

В этой главе:

- коротко об ООП;
- сборка библиотек классов;
- хранение данных в полях;
- запись и вызов методов;
- управление доступом с помощью свойств и индексаторов;
- сопоставление шаблонов с объектами;
- работа с записями.

Коротко об объектно-ориентированном программировании

Объект в реальном мире — это предмет, например автомобиль или человек. Объект в программировании часто представляет нечто в реальном мире, например товар или банковский счет, но может быть и чем-то более абстрактным.

В языке C# используются классы `class` (обычно) или структуры `struct` (редко) для определения каждого типа объекта. О разнице между классами и структурами вы узнаете в главе 6. Можно представить тип как шаблон объекта.

Ниже кратко описаны концепции объектно-ориентированного программирования.

- *Инкапсуляция* — комбинация данных и действий, связанных с объектом. К примеру, тип `BankAccount` может иметь такие данные, как `Balance` и `AccountName`, а также действия, такие как `Deposit` и `Withdraw`. При инкапсуляции часто возникает необходимость управлять тем, кто и что может получить доступ к этим действиям и данным, например ограничение доступа к внутреннему состоянию объекта или его изменению извне.
- *Композиция* — то, из чего состоит объект. К примеру, автомобиль состоит из разных частей, таких как четыре колеса, несколько сидений, двигатель и т. д.
- *Агрегирование* касается всего, что может быть объединено с объектом. Например, человек, не будучи частью автомобиля, может сидеть на водительском сиденье, а затем стать водителем. Два отдельных объекта объединены, чтобы сформировать новый компонент.
- *Наследование* — многократное использование кода с помощью подклассов, производных от базовых классов или суперклассов. Все функциональные возможности базового класса становятся доступными в производном классе. Например, базовый или суперкласс `Exception` имеет несколько членов, которые имеют одинаковую реализацию во всех исключениях. Подкласс же или производный класс `SqlException` наследует эти члены и имеет дополнительные, имеющие отношение только к тем случаям, когда возникает исключение базы данных SQL — например, свойство, содержащее информацию о подключении к базе данных.
- *Абстракция* — передача основной идеи объекта и игнорирование его деталей или особенностей. Язык C# имеет ключевое слово `abstract`, которое формализует концепцию. Если класс не явно абстрактный, то его можно описать как конкретный. Базовые классы часто абстрактны, например, суперкласс `Stream` — абстрактный, а его подклассы, такие как `FileStream` и `MemoryStream`, — конкретные. Абстракция — сложный баланс. Если вы сделаете класс слишком абстрактным, то большее количество классов сможет наследовать его, но количество возможностей для совместного использования уменьшится.
- *Полиморфизм* заключается в переопределении производным классом унаследованных методов для реализации собственного поведения.

Разработка библиотек классов

Сборки библиотек классов группируют типы в легко развертываемые модули (DLL-файлы). Не считая раздела, где вы изучали модульное тестирование, до сих пор вы создавали только консольные приложения, содержащие ваш код. Но чтобы он стал доступен для других проектов, его следует помещать в сборки библиотек классов, как это делают сотрудники корпорации Microsoft.

Создание библиотек классов

Первая задача — создать повторно используемую библиотеку классов .NET.

1. В существующей папке Code создайте папку Chapter05 с подпапкой PacktLibrary.
2. В программе Visual Studio Code выберите File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как), введите имя `Chapter05`, выберите папку Chapter05 и нажмите кнопку Save (Сохранить).
3. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PacktLibrary и нажмите кнопку Add (Добавить).
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

5. Откройте файл `PacktLibrary.csproj` и обратите внимание, что по умолчанию библиотеки классов нацелены на .NET 5 и, следовательно, могут работать только с другими сборками, совместимыми с .NET 5, как показано ниже в коде:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

6. Измените целевую платформу для поддержки .NET Standard 2.0, как показано ниже в коде:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

7. Сохраните и закройте файл.
8. На панели TERMINAL (Терминал) скомпилируйте проект, используя следующую команду: `dotnet build`.

Чтобы использовать новейшие функции языка C# и платформы .NET, поместите типы в библиотеку классов .NET 5. Для поддержки устаревших платформ .NET, таких как .NET Core, .NET Framework и Xamarin, поместите типы, которые повторно можно использовать, в библиотеку классов .NET Standard 2.0.

Определение классов

Следующая задача — определить класс, который будет представлять человека.

1. На панели EXPLORER (Проводник) переименуйте файл `Class1.cs` в `Person.cs`.
2. Щелкните кнопкой мыши на файле `Person.cs`, чтобы открыть его, и измените имя класса на `Person`.
3. Измените название пространства имен на `Packt.Shared`.

Мы делаем это, поскольку важно поместить ваши классы в логически именованное пространство имен. Лучшее имя пространства имен будет специфичным для домена, например System.Numerics для типов, связанных с расширенными числовыми функциями, но в нашем случае мы создадим типы Person, BankAccount и WondersOfTheWorld, и у них нет общего домена.

Ваш файл класса теперь должен выглядеть следующим образом:

```
using System;

namespace Packt.Shared
{
  public class Person
  {
  }
}
```

Обратите внимание, что ключевое слово `public` языка C# указывается перед словом `class`. Это ключевое слово называется *модификатором доступа*, управляющим тем, как осуществляется доступ к данному классу.

Если вы явно не определили доступ к классу с помощью ключевого слова `public` (публичный), то он будет доступен только в определяющей его сборке. Это следствие того, что неявный модификатор доступа для класса считается `internal` (внутренний). Нам же нужно, чтобы класс был доступен за пределами сборки, поэтому необходимо ключевое слово `public`.

Члены

У этого типа еще нет членов, инкапсулированных в него. Скоро мы их создадим. Членами могут быть поля, методы или специализированные версии их обоих. Их описание представлено ниже.

- *Поля* используются для хранения данных. Существует три специализированных категории полей:
  - *константы* — данные в них никогда не меняются. Компилятор буквально копирует данные в любой код, который их читает;

  - *поля, доступные только для чтения,* — данные в таких полях не могут измениться после создания экземпляра класса, но могут быть рассчитаны или загружены из внешнего источника во время создания экземпляра;
  - *события* — данные ссылаются на один или несколько методов, вызываемых автоматически при возникновении определенной ситуации, например при нажатии кнопки. Тема событий будет рассмотрена в главе 6.
- *Методы* используются для выполнения операторов. Вы уже ознакомились с некоторыми примерами в главе 4. Существует четыре специализированных метода:
  - *конструкторы* выполняются, когда вы используете ключевое слово `new` для выделения памяти и создания экземпляра класса;
  - *свойства* выполняются, когда необходимо получить доступ к данным. Они обычно хранятся в поле, но могут храниться извне или рассчитываться во время выполнения. Использование свойств — предпочтительный способ инкапсуляции полей, если только не требуется выдать наружу адрес памяти поля;
  - *индексаторы* выполняются, когда необходимо получить доступ к данным с помощью синтаксиса массива `[]`;
  - *операции* выполняются, когда необходимо применить операции типа `+` и `/` для операндов вашего типа.

Создание экземпляров классов

В этом подразделе мы создадим *экземпляр* класса `Person` (данный процесс описывается как *инстанцирование* класса).

Ссылка на сборку

Прежде чем мы сможем создать экземпляр класса, нам нужно сослаться на сборку, которая его содержит.

1. Создайте подпапку PeopleApp в папке Chapter05.
2. В программе Visual Studio Code выберите File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PeopleApp и нажмите кнопку Add (Добавить).
3. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите пункт `PeopleApp`.
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

5. На панели EXPLORER (Проводник) щелкните кнопкой мыши на файле `PeopleApp.csproj`.
6. Добавьте ссылку на проект в `PacktLibrary`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
  </ItemGroup>
</Project>
```

7. На панели TERMINAL (Терминал) введите команду для компиляции проекта `PeopleApp` и его зависимого проекта `PacktLibrary`, как показано в следующей команде:

```
dotnet build
```

8. Выберите `PeopleApp` в качестве активного проекта для OmniSharp.

Импорт пространства имен для использования типа

Теперь мы готовы написать операторы для работы с классом `Person`.

1. В программе Visual Studio Code в папке PeopleApp откройте проект `Program.cs`.
2. В начале файла `Program.cs` введите операторы для импорта пространства имен для нашего класса `People` и статически импортируйте класс `Console`, как показано ниже:

```
using Packt.Shared;
using static System.Console;
```

3. В методе `Main` введите операторы для:
   - создания экземпляра типа `Person`;
   - вывода экземпляра, используя его текстовое описание.

   Ключевое слово `new` выделяет память для объекта и инициализирует любые внутренние данные. Мы могли бы использовать `Person` вместо ключевого слова `var`, но применение последнего требует меньше ввода и по-прежнему понятно, как показано ниже:

```
var bob = new Person();
WriteLine(bob.ToString());
```

Вы можете спросить: «Почему у переменной bob имеется метод ToString? Класс Person пуст!» Не беспокойтесь, скоро вы все узнаете!

4. Запустите приложение, введя команду dotnet run на панели TERMINAL (Терминал), а затем проанализируйте результат:

```
Packt.Shared.Person
```

Управление несколькими файлами

Если требуется одновременная работа с несколькими файлами, то вы можете размещать их рядом друг с другом по мере их редактирования.

1. На панели EXPLORER (Проводник) разверните два проекта.
2. Откройте файлы Person.cs и Program.cs.
3. Нажав и удерживая кнопку мыши, перетащите вкладку окна редактирования для одного из ваших открытых файлов, чтобы расположить их так, чтобы вы могли одновременно видеть оба файла Person.cs и Program.cs.

Вы можете нажать кнопку Split Editor Right (Разделить редактор) или нажать сочетание клавиш Ctrl+\ или Cmd+\, чтобы разместить два окна файла друг рядом с другом.

Более подробно о работе с пользовательским интерфейсом Visual Studio Code вы можете прочитать на сайте: https://code.visualstudio.com/docs/getstarted/userinterface.

Работа с объектами

Хотя наш класс Person явно не наследуется ни от какого типа, все типы косвенно наследуются от специального типа System.Object. Реализация метода ToString в типе System.Object выдает полные имена пространства имен и типа.

Возвращаясь к исходному классу Person, мы могли бы явно сообщить компилятору, что Person наследуется от типа System.Object:

```
public class Person : System.Object
```

Когда класс Б *наследуется* от класса А, мы говорим, что А — *базовый класс* или суперкласс, а Б — *производный класс*. В нашем случае System.Object — базовый класс (суперкласс), а Person — производный.

Мы также можем использовать в C# псевдоним типа — ключевое слово object:

```
public class Person : object
```

Наследование System.Object

Сделаем наш класс явно наследуемым от `object`, а затем рассмотрим, какие члены имеют все объекты.

1. Измените свой класс `Person` для явного наследования от `object`.
2. Затем установите указатель мыши внутри ключевого слова `object` и нажмите клавишу F12 или щелкните правой кнопкой мыши на ключевом слове `object` и выберите Go to Definition (Перейти к определению).

Вы увидите определение типа `System.Object` и его членов. Вам не нужно разбираться во всем этом определении, но обратите внимание на метод `ToString`, показанный на рис. 5.1.

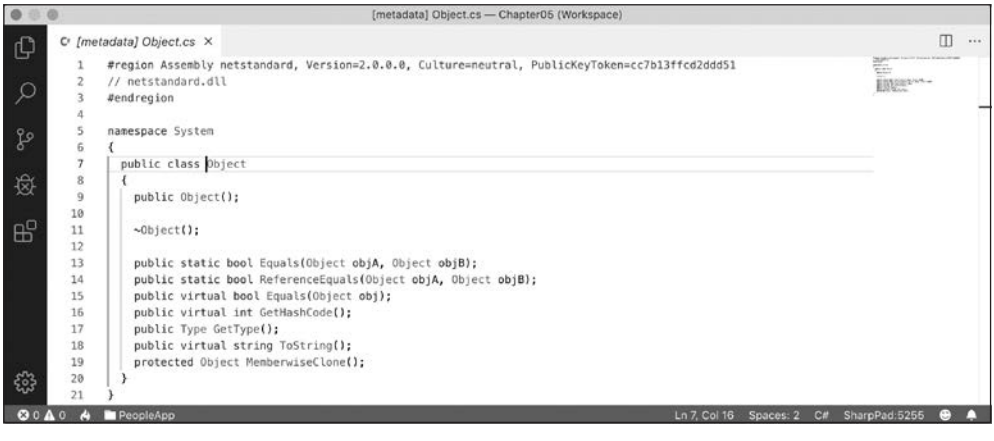

Рис. 5.1. Определение класса System.Object

Будьте уверены: другие программисты знают, что, если наследование не указано, класс наследуется от System.Object.

Хранение данных в полях

Теперь определим в классе несколько полей для хранения информации о человеке.

Определение полей

Допустим, мы решили, что информация о человеке включает имя и дату рождения. Мы инкапсулируем оба значения в классе `Person` и также сделаем поля общедоступными.

В классе `Person` напишите операторы для объявления двух общедоступных полей для хранения имени и даты рождения человека:

```
public class Person : object
{
  // поля
  public string Name;
  public DateTime DateOfBirth;
}
```

Вы можете использовать любые типы для полей, включая массивы и коллекции, такие как списки и словари. Они вам будут полезны при необходимости хранить несколько значений в одном именованном поле. В данном примере информация о человеке содержит только одно имя и одну дату рождения.

Модификаторы доступа

При реализации инкапсуляции важно выбрать, насколько видны элементы.

Обратите внимание: работая с классом, мы использовали ключевое слово `public` по отношению к созданным полям. Если бы мы этого не сделали, то поля были бы закрытыми, то есть доступными только в пределах класса.

Доступны четыре ключевых слова для модификаторов доступа, каждое из которых вы можете применить к члену класса, например к полю или методу, как показано в табл. 5.1.

Таблица 5.1. Модификаторы доступа

Модификатор доступа	Описание
private	Доступ ограничен содержащим типом. Используется по умолчанию
internal	Доступ ограничен содержащим типом и любым другим типом в текущей сборке
protected	Доступ ограничен содержащим типом или типами, производными от содержащего типа
public	Неограниченный доступ
internal protected	Доступ ограничен содержащим типом и любым другим типом в текущей сборке, а также типами, производными от содержащего класса. Аналогичен вымышленному модификатору доступа internal_or_protected (то есть дает доступ по принципу «internal ИЛИ protected»).
private protected	Доступ ограничен содержащим типом и любым другим типом, который наследуется от типа и находится в той же сборке. Аналогичен вымышленному модификатору доступа internal_and_protected (то есть дает доступ по принципу «internal И protected»). Эта комбинация доступна только для версии C# 7.2 или более поздних