# 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 с помощью полей;
- запись и вызов методов;
- управление доступом с помощью свойств и индексаторов;
- сопоставление с образцом с помощью объектов;
- 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 ограничивать доступ к внутреннему состоянию объекта или его изменению извне.
- *Композиция* — то, из чего состоит объект. К примеру, объект `Car` (Автомобиль) состоит из разных частей, таких как четыре объекта `Wheel` (Колесо), несколько объектов `Seat` (Сиденье) и один объект `Engine` (Двигатель).
- *Агрегирование* касается всего, что может быть объединено с объектом. Например, объект `Person` не является частью объекта `Car`, но человек может сесть на сиденье водителя (`Seat`), а затем стать им (`Driver`). Два отдельных объекта объединены, чтобы сформировать новый компонент.
- *Наследование* — многократное использование кода с помощью *подкласса*, производного от *базового класса* или *суперкласса*. Все функциональные возможности базового класса наследуются и становятся доступными в *производном* классе. Например, базовый или суперкласс `Exception` имеет несколько членов, которые имеют одинаковую реализацию во всех исключениях. А подкласс, он же 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. Создавать объекты можно только с помощью конкретных классов; абстрактные же могут использоваться лишь в качестве основы для других классов, поскольку им не хватает некоторой реализации. Абстракцию сложно сбалансировать. Если вы сделаете класс слишком абстрактным, то больше классов смогут наследовать его, но при этом уменьшится количество возможностей для совместного использования.
- *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 или блокноты .NET Interactive, содержащие ваш код. Но, чтобы он стал доступен для нескольких проектов, его следует помещать в сборки библиотек классов, как это делает Microsoft.

## Создание библиотек классов

Первая задача — создать повторно используемую библиотеку классов .NET.

1. Откройте редактор кода и создайте библиотеку классов с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `Chapter05`;
   3) файл и папка проекта: `PacktLibrary`.
2. Откройте файл `PacktLibrary.csproj` и обратите внимание, что по умолчанию библиотеки классов нацелены на .NET 6 и, следовательно, могут работать только с другими сборками, совместимыми с .NET 6, как показано ниже:

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

3. Измените целевую платформу для поддержки .NET Standard 2.0 и удалите записи, допускающие неявное использование и значение NULL:

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

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

</Project>
```

4. Сохраните и закройте файл.
5. Удалите файл `Class1.cs`.

6. Скомпилируйте проект, чтобы другие проекты позже могли на него ссылаться:
   1) в Visual Studio Code введите следующую команду: `dotnet build`;
   2) в Visual Studio выберите команду меню Build ▸ Build PacktLibrary (Сборка ▸ Создать PacktLibrary).

Чтобы использовать новейшие функции языка C# и платформы .NET, поместите типы в библиотеку классов .NET 6. 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 в пространстве имен

Следующая задача — определить класс, который будет представлять человека.

1. Добавьте новый файл класса `Person.cs`.
2. Статически импортируйте `System.Console`.
3. Установите пространство имен `Packt.Shared`.

Мы делаем это, поскольку важно поместить ваши классы в логически названное пространство имен. Лучшее название пространства имен будет специфичным для домена, например System.Numerics для типов, связанных с расширенными числами. В нашем случае мы создадим типы Person, BankAccount и WondersOfTheWorld, и у них нет общего домена, поэтому мы будем использовать более общий Packt.Shared.

Ваш файл класса теперь должен выглядеть следующим образом:

```
using System;
using static System.Console;

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

Обратите внимание, что ключевое слово `public` языка C# указывается перед классом. 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`, то он будет доступен только в определяющей его сборке. Это следствие того, что неявный модификатор доступа для класса считается `internal` (внутренним). Нам же нужно, чтобы класс был доступен за пределами сборки, поэтому необходимо ключевое слово `public`.

### Упрощение объявлений пространств имен

Если вы работаете в .NET 6.0 и, следовательно, используете C# 10 или более позднюю версию, то можете завершить объявление пространства имен точкой с запятой и удалить фигурные скобки, чтобы упростить код:

```
using System;

namespace Packt.Shared; // класс в этом файле находится
                        // в данном пространстве имен

public class Person
{
}
```

Описанное выше также известно как объявление пространства имен в файловой области. У вас может быть только одно пространство имен в файловой области для каждого файла. Далее в главе мы будем использовать это в библиотеке классов, предназначенной для .NET 6.0.

Поместите каждый новый тип в отдельный файл, чтобы можно было использовать объявления пространств имен в файловой области.

## 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 специализированные 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 них 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 специализированные категории методов:
  - *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`.

### Ссылка на сборку

Прежде чем мы сможем создать экземпляр класса, нам нужно сослаться на сборку, которая его содержит. Мы будем использовать класс в консольном приложении.

1. Откройте редактор кода и создайте консольное приложение `PeopleApp` в рабочей области/решении `Chapter05`.
2. Если вы используете Visual Studio Code:
   1) выберите `PeopleApp` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку `Yes` (Да), чтобы добавить их;
   2) измените файл `PeopleApp.csproj`, чтобы добавить ссылку на проект в `PacktLibrary`, как показано ниже (выделено жирным шрифтом):

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
```

```
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
  </ItemGroup>

</Project>
```

3) на панели TERMINAL (Терминал) введите команду для компиляции проекта PeopleApp и его зависимого проекта PacktLibrary:

```
dotnet build
```

3. Если вы используете Visual Studio:
   1) настройте стартовый проект для решения в соответствии с текущим выбором;
   2) на панели Solution Explorer (Обозреватель решений) выберите проект PeopleApp, выберите команду меню Project ▸ Add Project Reference (Проект ▸ Добавить ссылку на проект), установите флажок, чтобы выбрать проект PacktLibrary, и нажмите кнопку OK;
   3) выберите команду меню Build ▸ Build PeopleApp (Сборка ▸ Создать PeopleApp).

## Импорт пространства имен для использования типа

Теперь мы готовы написать операторы для работы с классом Person.

1. В проекте/папке PeopleApp откройте файл Program.cs.
2. В начале файла Program.cs удалите комментарий и введите следующие операторы, чтобы импортировать пространство имен для нашего класса Person и статически импортировать класс Console:

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

3. В файле Program.cs введите операторы:
   - для создания экземпляра типа Person;
   - для вывода экземпляра с помощью его текстового описания.

   Ключевое слово new выделяет память для объекта и инициализирует любые внутренние данные. Мы могли бы использовать var вместо имени класса Person, но тогда нам нужно было бы указать Person после ключевого слова new:

```
// Person bob = new Person(); // C# 1.0 или более поздние версии
// var bob = new Person(); // C# 3.0 или более поздние версии
Person bob = new(); // C# 9.0 или более поздние версии
WriteLine(bob.ToString());
```

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

4. Запустите код и проанализируйте результат:

```
Packt.Shared.Person
```

# Работа с объектами

Хотя наш класс Person явно не наследуется ни от какого типа, все типы в конечном счете прямо или 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 (подкласс). В нашем случае System.Object — базовый класс (суперкласс), а Person — производный (подкласс).

Мы также можем использовать псевдоним C# — 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, чтобы он мог явно наследовать 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 определенные Microsoft тип System.Object и его члены. Вам пока не нужно детально разбираться во всем этом, но обратите внимание на метод ToString, показанный на рис. 5.1.

Исходите из предположения, что другие программисты знают о том, что если наследование не указано, то класс наследуется от System.Object.

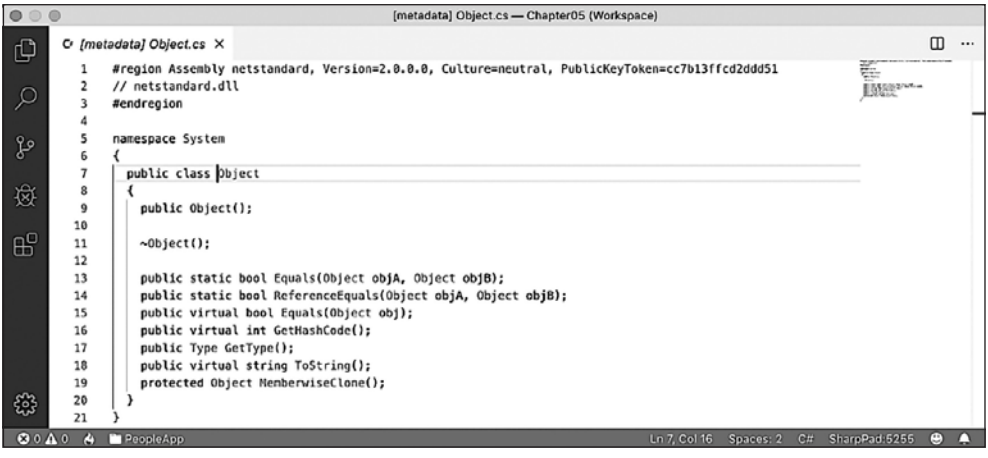


**Рис. 5.1.** Определение класса System.Object

# 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 сделаем поля общедоступными.

В классе `Person` напишите операторы, чтобы объявить два общедоступных поля 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. С их помощью вы сможете 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 степень видимости членов.

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.

Существует четыре ключевых слова для модификаторов доступа и две их комбинации. Вы можете применить их 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 и любым другим типом, который наследуется от него |
| `public` | Неограниченный доступ |
| `internal protected` | Доступ ограничен содержащим типом и любым другим типом в текущей сборке, а также любым другим типом, который наследуется от содержащего. Аналогичен вымышленному модификатору доступа `internal_or_protected` |
| `private protected` | Доступ ограничен содержащим типом и любым другим типом, который наследуется от него и находится в той же сборке. Аналогичен вымышленному модификатору доступа `internal_and_protected`. Эта комбинация доступна только для версии C# 7.2 или более поздних |

Следует явно применять один из модификаторов доступа ко всем членам типа, даже если будет использоваться модификатор по умолчанию, именуемый private. Кроме того, поля, как правило, должны быть private или protected, и вам следует создать общедоступные свойства public, чтобы получить или установить значения полей, поскольку так можно контролировать доступ. Вы сделаете это позже в данной главе.

## Установка и вывод значений полей

Теперь мы будем использовать эти поля в вашем коде.

1. Убедитесь, что в начале файла `Program.cs` импортировано пространство имен `System`. Нам необходимо сделать это, чтобы использовать тип `DateTime`.

2. После того как экземпляр bob будет создан, добавьте следующие операторы, чтобы задать имя и дату рождения человека, а затем выведите эти поля в надлежащем формате:

```
bob.Name = "Bob Smith";
bob.DateOfBirth = new DateTime(1965, 12, 22); // C# 1.0 или более
                                              // поздние версии

WriteLine(format: "{0} was born on {1:dddd, d MMMM yyyy}",
  arg0: bob.Name,
  arg1: bob.DateOfBirth);
```

Мы могли бы также использовать интерполяцию строк, но длинные строки будут переноситься на несколько строк, что может усложнить чтение кода в данной книге. Обратите внимание: в примерах кода, описанных в книге, {0} — это заполнитель для arg0 и т. д.

3. Запустите код и проанализируйте результат:

```
Bob Smith was born on Wednesday, 22 December 1965
```

Помните, что вывод может выглядеть по-разному в зависимости от вашего региона, то есть языка и культуры.

Код формата для arg1 состоит из нескольких частей: dddd означает день недели, d — день месяца, MMMM — название месяца. Строчные буквы m используются для значений времени в минутах, yyyy означает полное число года, yy — год, обозначенный двумя цифрами.

Вы также можете инициализировать поля, используя сокращенный синтаксис *инициализатора объекта* с фигурными скобками. Рассмотрим пример.

4. Добавьте операторы ниже существующего кода, чтобы создать запись еще об одном человеке, женщине по имени Alice. Обратите внимание на иной формат кода даты рождения при записи в консоль:

```
Person alice = new()
{
  Name = "Alice Jones",
  DateOfBirth = new(1998, 3, 7) // C# 9.0 и более поздние версии
};

WriteLine(format: "{0} was born on {1:dd MMM yy}",
  arg0: alice.Name,
  arg1: alice.DateOfBirth);
```

5. Запустите код и проанализируйте результат:

```
Alice Jones was born on 07 Mar 98
```